Incremental Detection of Inconsistencies in Distributed Data

Wenfei Fan[1,2] Jianzhong Li[2] Nan Tang[3] Wenyuan Yu[1]

[1]*University of Edinburgh* [2]*Harbin Institute of Technology* [3]*QCRI, Qatar Foundation*
wenfei@inf.ed.ac.uk lijzh@hit.edu.cn ntang@qf.org.qa wenyuan.yu@ed.ac.uk

Abstract—This paper investigates incremental detection of errors in distributed data. Given a distributed database D, a set Σ of conditional functional dependencies (CFDs), the set V of violations of the CFDs in D, and updates ΔD to D, it is to find, with minimum data shipment, changes ΔV to V in response to ΔD. The need for the study is evident since real-life data is often dirty, distributed and frequently updated. It is often prohibitively expensive to recompute the entire set of violations when D is updated. We show that the incremental detection problem is NP-complete for database D that is partitioned either vertically or horizontally, even when Σ and D are fixed. Nevertheless, we show that it is *bounded*: there exist algorithms to detect errors such that their computational cost and data shipment are both *linear* in the size of ΔD and ΔV, *independent of* the size of the database D. We provide such incremental algorithms for vertically partitioned data and horizontally partitioned data, and show that the algorithms are optimal. We further propose optimization techniques for the incremental algorithm over vertical partitions to reduce data shipment. We verify experimentally, using real-life data on Amazon Elastic Compute Cloud (EC2), that our algorithms substantially outperform their batch counterparts.

Index Terms–Incremental Algorithms; Distributed Data; Conditional Functional Dependencies; Error Detection.

1 INTRODUCTION

Real-life data is often dirty. To clean the data, efficient algorithms for detecting errors have to be in place. Errors in the data are typically detected as violations of constraints (data quality rules), such as functional dependencies (FDs), denial constraints [3], and conditional functional dependencies (CFDs) [9]. When the data is in a centralized database, it is known that two SQL queries suffice to detect its violations of a set of CFDs [9].

It is increasingly common to find data *partitioned* vertically (*e.g.,* [29]) or horizontally (*e.g.,* [18]), and *distributed* across different sites. This is highlighted by the recent interests in SaaS and Cloud computing, MapReduce [7], [24] and columnar DBMS [29]. In the distributed settings, however, it is much harder to detect errors in the data.

Example 1: Consider an employee relation D_0 shown in Fig. 2, which consists of tuples t_1–t_5 (ignore t_6 for the moment), and is specified by the following schema:

EMP(id, name, sex, grade, street, city, zip, CC, AC, phn, salary, hd)

Each EMP tuple specifies the id, name, sex, salary grade level, address (street, city, zip code), phone number (country code CC, area code AC, phone phn), salary and the date hired (hd). Here the employee id is a *key* of EMP.

CFDs	Violations
ϕ_1 : ([CC = 44, zip] $\rightarrow$ [street])	t_1, t_3, t_4, t_5
ϕ_2 : ([CC = 44, AC = 131] $\rightarrow$ [city = 'EDI'])	t_1

Fig. 1. Example CFDs and their violations

To detect errors, a set of CFDs is defined on the EMP relation, as shown in Fig. 1. Here ϕ_1 asserts that for employees in the UK (*i.e.,* CC = 44), zip code uniquely determines street. CFD ϕ_2 assures that for any UK employee, if the area code is 131 then the city must be EDI.

Errors in D_0 emerge as violations of the CFDs, *i.e.,* those tuples in D_0 that violate at least one CFD in Σ_0, as shown in Fig. 1. For instance, t_1 and t_5 violate ϕ_1: they represent UK employees with the same zip, but have different street's. Moreover, t_1 *alone* violates ϕ_2: t_1[CC] = 44 and t_1[AC] = 131, but t_1[city] = 'NYC' $\neq$ 'EDI'. When D_0 is in a centralized database, the violations can be easily caught by using SQL-based techniques [9].

Now consider distributed settings. As depicted in Fig. 2, D_0 is partitioned either (1) vertically into three fragments D_{V_1}, D_{V_2} (grey columns) and D_{V_3}, all with attribute id; or (2) horizontally into D_{H_1} (t_1–t_2), D_{H_2} (t_3–t_4) and D_{H_3} (t_5), for employees with salary grade 'A' (junior level), 'B' and 'C' (senior), respectively. The fragments are distributed over different sites.

To find violations in both settings, it is necessary to *ship data from one site to another*. For instance, to find the violations of ϕ_1 in the vertical partitions, one has to send tuples with CC = 44 from the site of D_{V_3} to the site of D_{V_2}, or the other way around to ship attributes (street, zip); similarly for the horizontal partitions. □

It is NP-complete to find violations of CFDs, with minimum data shipment, in a distributed relation that is partitioned either horizontally or vertically [10]. A heuristic algorithm was developed in [10] to compute the violations of CFDs in *horizontally* partitioned data, which takes 80 seconds to find violations of one CFD in 8 fragments (*i.e.,* 8 sites) of 1.6 million tuples.

Distributed data is also typically *dynamic*, *i.e.,* frequently updated [25]. It is often prohibitively expensive to recompute the entire violations in a distributed

		D_{V_1}				D_{V_2}(with id replica)			D_{V_3}(with id replica)					Updates
		id	name	sex	grade	street	city	zip	CC	AC	phn	salary	hd	
D_{H_1}	t_1 :	1	Mike	M	A	Mayfield	NYC	EH4 8LE	44	131	8693784	65k	01/10/2005	
	t_2 :	2	Sam	M	A	Preston	EDI	EH2 4HF	44	131	8765432	65k	01/05/2009	
D_{H_2}	t_3 :	3	Molina	F	B	Mayfield	EDI	EH4 8LE	44	131	3456789	80k	01/03/2010	
	t_4 :	4	Philip	M	B	Mayfield	EDI	EH4 8LE	44	131	2909209	85k	01/05/2010	delete
D_{H_3}	t_5 :	5	Adam	M	C	Crichton	EDI	EH4 8LE	44	131	7478626	120k	01/05/1995	
	t_6 :	6	George	M	C	Mayfield	EDI	EH4 8LE	44	131	9595858	120k	01/07/1993	insert

Fig. 2. An EMP relation D_0

database D when D is updated. This motivates us to study *incremental detection* of errors. In a nutshell, let V denote the violations of a set Σ of CFDs in D, ΔD be updates to D, and $D \oplus \Delta D$ denote the database updated by ΔD. In contrast to *batch algorithms* that compute violations of Σ in D starting from scratch, incremental detection is to find *changes* $\Delta\mathsf{V}$ to V, which aims to minimize unnecessary recomputation. Indeed, when ΔD is small, $\Delta\mathsf{V}$ is often small as well, though $\Delta\mathsf{V}$ may include tuples from ΔD and D. It is more efficient to compute $\Delta\mathsf{V}$ than the entire violations of Σ in $D \oplus \Delta D$.

Example 2: Consider ϕ_1 of Fig. 1, relation D_0 and its partitions given in Fig. 2, and the updates below.

(1) Insertions. Assume that t_6 is inserted into D_0, as shown in Fig. 2. Then the new violation $\Delta\mathsf{V}$ is $\{t_6\}$.

(a) *Batch computation*. In the vertical partitions, one needs to ship either tuples with the same $(\mathsf{zip}, \mathsf{street})$ as t_6 (in D_{V_2}) or 6 tuples with CC = 44 (D_{V_3}), as shown in Example 1. In the horizontal partition, we have to compare all tuples with CC = 44, which requires the shipment of 4 (partial) tuples.

(b) *Incremental computation*. Since t_5 is already a violation of ϕ_1 in V and (t_5, t_6) together violate ϕ_1, we can conclude that t_6 is the only new violation of ϕ_1, *i.e.*, $\Delta\mathsf{V} = \{t_6\}$ for ϕ_1. Indeed, for any tuple t, if (t, t_6) violate ϕ_1, then either (t, t_5) violate ϕ_1 or $t[\mathsf{CC}, \mathsf{zip}, \mathsf{street}] = t_5[\mathsf{CC}, \mathsf{zip}, \mathsf{street}]$. In both cases, t is already in V (*i.e.*, a violation). Hence to find $\Delta\mathsf{V}$ for ϕ_1, one needs to ship a single tuple id in the vertical partition (Section 4), and no data to be shipped in the horizontal case (Section 6).

(2) Deletions. Assume that t_4 is deleted after the insertion of t_6. One can verify that only t_4 has to be removed from the violations of ϕ_1, *i.e.*, $\Delta\mathsf{V} = \{t_4\}$ for ϕ_1.

(a) *Batch computation*. To find violations of ϕ_1 in $D_0 \oplus \Delta D$, one has to ship the same amount of data as in (1)(a).

(b) *Incremental computation*. In contrast, since t_3, t_4 are both in V and $t_3[\mathsf{street}, \mathsf{zip}] = t_4[\mathsf{street}, \mathsf{zip}]$, one can verify that only t_4 should be removed from V. Indeed, for any t, if (t, t_4) violate ϕ_1, so do (t, t_3). Since t_3 remains in V, so does t. Again, one needs to ship a single tuple id in vertical partitions, and no data in the horizontal case. □

It has been verified in a number of applications that incremental algorithms are more efficient than their batch counterparts when updates are small [26]. This example shows that this holds for distributed error detection.

Contributions. This paper establishes the complexity bounds and provides efficient algorithms for incrementally detecting the violations of CFDs in fragmented and distributed data, either vertically or horizontally.

(1) We formulate incremental detection as an optimization problem, and establish its complexity bounds (Section 3). We show that the problem is NP-complete even when both D and CFDs are fixed, *i.e.*, when only the size $|\Delta D|$ of updates varies. Nevertheless, we show that the problem is *bounded* [27]: there exist algorithms for incremental detection such that their communication costs and computational costs are functions in the size of *the changes* in the input and output (*i.e.*, $|\Delta D|$ and $|\Delta\mathsf{V}|$), *independent of the size of database D*. This tells us that incremental detection can be carried out efficiently, since in practice, ΔD and $\Delta\mathsf{V}$ are typically small.

(2) We develop an algorithm for incrementally detecting violations of CFDs for vertical partitions (Section 4). We show that the algorithm is *optimal* [27]: both its communication costs and computational costs are *linear* in $|\Delta D|$ and $|\Delta\mathsf{V}|$. Indeed, $|\Delta D|$ and $|\Delta\mathsf{V}|$ characterize the amount of work that is *absolutely necessary* to perform for incremental detection [27].

(3) We develop optimization methods (Section 5) to further reduce data shipment for error detection in vertical partitions. The idea is to identify and maximally share indices among CFDs such that when multiple CFDs demand the shipment of the same tuples, only a single copy of the data is shipped. We show that the problem for building optimal indices is NP-complete, but provide an efficient heuristic algorithm.

(4) We also provide an incremental detection algorithm for horizontal partitions (Section 6). We show that the algorithm is also *optimal*, as for its vertical counterpart.

(5) Using TPCH for large scale data and DBLP for real-life data, we conduct experiments on Amazon EC2. We find that our incremental algorithms outperform their batch counterparts by *two orders of magnitude*, for fairly large updates (up to 10GB for TPCH). Moreover, our methods scale well with both the size of data and the number of CFDs. We also find the optimization strategies effective.

This work provides fundamental results and a practical solution for error detection in distributed data. We focus on CFDs because they carry constant patterns and are difficult to handle, and moreover, as shown in [9], they capture inconsistencies that traditional dependencies fail to catch. The techniques developed here, nonetheless, can be readily used to incrementally detect violations of other dependencies used in data cleaning, such as functional dependencies and denial constraints.

We discuss related work below, review error detection in distributed data in Section 2, and conclude in Section 8.

Related work. This work extends [11] by including (1) detailed proofs of the fundamental problems in connection with incremental error detection (Section 3); (2) a proof of the intractability of the optimization problem for vertical partitions (Section 4); (3) an optimal algorithm for horizontal partitions (Section 6); and (4) its experimental study (Section 7). Neither (3) nor (4) was studied in [11]. Proofs of (1) and (2) were not presented in [11].

Methods for (incrementally) detecting CFD violations are studied in [9] for centralized data, based on SQL techniques. There has been work on constraint enforcement in distributed databases (*e.g.,* [2], [16], [17]). As observed in [16], [17], constraint checking is hard in distributed settings, and hence, certain conditions are imposed there so that their constraints can be checked locally at individual site, without data shipment. As shown by the examples above, however, to find CFD violations it is often necessary to ship data. Detecting constraint violations has been studied in [2] for monitoring distributed systems, which differs substantially from this work in that their constraints are defined on *system states* and cannot express CFDs. In contrast, CFDs are to detect errors in *data*, which is typically much larger than system states. Closer to this work is [10], which studies CFD violation detection in horizontal partitions, but considers neither incremental detection nor algorithms for detecting errors in vertical partitions.

Incremental algorithms have proved useful in a variety of areas (see [26] for a survey). In particular, incremental view maintenance has been extensively studied [14], notably for distributed data [4], [6], [15], [28]. Various auxiliary structures have been proposed to reduce data shipment, *e.g.,* counters [6], [15], pointer [28] and tags in base relations [4]. While these could be incorporated into our solution, they do not yield bounded/optimal incremental detection algorithms.

There has also been a host of work on query processing [20] and multi-query optimization [19] for distributed data. The former typically aims to generate distributed query plans, to reduce data shipment or response time (see [20] for a survey). Optimization strategies, *e.g.,* semiJoins [5], bloomJoins [22], and recently [8], [21], [23], [30], have proved useful in main-memory distributed databases (*e.g.,* MonetDB [12] and H-Store [18]), and in cloud computing and MapReduce [7], [24]. Our algorithms leverage the techniques of [19] to reduce data shipment when validating multiple CFDs, in particular.

2 Error Detection in Distributed Data

In this section we review CFDs [9], data fragmentation [25] and error detection in distributed data [10].

2.1 Conditional Functional Dependencies

A CFD ϕ on relation R is a pair ($X \rightarrow Y$, t_p), where (1) $X \rightarrow Y$ is a standard functional dependency (FD) on R; and (2) t_p is the *pattern tuple* of ϕ with attributes in X and Y, where for each attribute A in $X \cup Y$, $t_p[A]$ is either a constant in the domain $\mathsf{dom}(A)$ of A, or an unnamed variable '_' that draws values from $\mathsf{dom}(A)$ [25].

Example 3: The CFDs in Fig. 1 can be expressed as:

ϕ_1: ([CC, zip] $\rightarrow$ [street], $\quad t_{p_1}$ = (44, _, _))
ϕ_2: ([CC, AC] $\rightarrow$ [city], $\quad t_{p_2}$ = (44, 131, EDI))

Note that FDs are a special case of CFDs in which the pattern tuple consists of '_' only. □

To give the semantics of CFDs, we use an operator $\asymp$ defined on constants and '_': $v_1 \asymp v_2$ if either $v_1 = v_2$, or one of v_1, v_2 is '_'. The operator extends to tuples, *e.g.,* (131, EDI) $\asymp$ (_, EDI) but (131, EDI) $\not\asymp$ (_, NYC).

An instance D of R *satisfies* a CFD ϕ, denoted by $D \models \phi$, iff for *all* tuples t and t' in D, if $t[X] = t'[X] \asymp t_p[X]$, then $t[Y] = t'[Y] \asymp t_p[Y]$. Intuitively, ϕ is defined on those tuples t in D such that $t[X]$ matches the pattern $t_p[X]$, and moreover, it enforces the pattern $t_p[Y]$ on $t[Y]$.

Example 4: Consider D_0 in Fig. 2 and the CFDs in Fig. 1. Then D_0 does not satisfy ϕ_1, since $t_1[\mathsf{CC}, \mathsf{zip}] = t_5[\mathsf{CC}, \mathsf{zip}]$ $\asymp$ (44, _) but $t_1[\mathsf{street}] \neq t_5[\mathsf{street}]$, violating ϕ_1. □

A set of CFDs of the form $(X \rightarrow Y, t_{p_i})$ $(i \in [1, n])$ can be converted to an equivalent form $(X \rightarrow Y, T_p)$, where T_p is a pattern tableau that contains n tuples $t_{p_1}, \cdots, t_{p_n}$ [9]. This is what we used in our implementation.

We call $(X \rightarrow B, t_p)$ a *constant* CFD if $t_p[B]$ is a constant, and a *variable* CFD if $t_p[B]$ is '_'. For instance, ϕ_2 in Fig. 1 is a constant CFD, while ϕ_1 is a variable CFD.

2.2 Data Fragmentation

We consider relations D of schema R that are partitioned into fragments, either vertically or horizontally.

Vertical partitions. In some applications (*e.g.,* [29]) one wants to partition D into $(D_1, \ldots, D_n)$ [25] such that

$$D_i = \pi_{X_i}(D), \qquad D = \bowtie_{i \in [1,n]} D_i,$$

where X_i is a set of attributes of R on which D is projected, including a *key* attribute of R. Relation D can be reconstructed by join operations on the *key* attribute.

Each vertical fragment D_i has its own schema R_i with attributes X_i. The set of attributes of R is $\bigcup_{i \in [1,n]} X_i$.

As shown in Fig. 2, D_0 can be partitioned vertically into D_{V_1}, D_{V_2} and D_{V_3}, where the schema of D_{V_1} is R_1(id, name, sex and grade); similarly for D_{V_2} and D_{V_3}.

Horizontal partitions. Relation D may also be partitioned (fragmented) into (D_1, ..., D_n) [18], [25] such that

$$D_i = \sigma_{F_i}(D), \qquad D = \bigcup_{i \in [1,n]} D_i,$$

where F_i is a Boolean predicate and selection $\sigma_{F_i}(D)$ identifies fragment D_i. These fragments are disjoint, *i.e.,* no tuple t appears in distinct fragments D_i and D_j $(i \neq j)$. They have the same schema R. The original relation D can be reconstructed by the union of these fragments.

For example, D_0 is horizontally partitioned into D_{H_1}, D_{H_2} and D_{H_3} in Fig. 2, with the selection predicate as grade = 'A', grade = 'B' and grade = 'C', respectively.

2.3 Detecting CFD Violations in Distributed Data

When CFDs are used as data quality rules, errors in the data are captured as violations of CFDs [9], [10].

Violations. For a CFD $\phi = (X \to Y, t_p)$ and an instance D of R, we use $\mathsf{V}(\phi, D)$ to denote the set of all tuples in D that violate ϕ, called the *violations of* ϕ *in* D. Here a tuple $t \in \mathsf{V}(\phi, D)$ iff there exists $t' \in D$ such that $t[X] = t'[X] \asymp t_p[X]$ but either $t[Y] \neq t'[Y]$ or $t[Y] = t'[Y] \not\asymp t_p[Y]$. For a set Σ of CFDs, we define $\mathsf{V}(\Sigma, D) = \bigcup_{\phi \in \Sigma} \mathsf{V}(\phi, D)$.

For instance, Fig. 1 lists violations of ϕ_1 and ϕ_2 in D_0.

When D is a centralized database, two SQL queries suffice to find $\mathsf{V}(\Sigma, D)$, no matter how many CFDs are in Σ. The SQL queries can be automatically generated [9].

Error detection in distributed data. Now consider a relation D that is partitioned into fragments $(D_1, \ldots, D_n)$, either vertically or horizontally. Assume *w.l.o.g.* that D_i's are distributed across distinct sites, *i.e.,* D_i resides at site S_i for $i \in [1, n]$, and S_i and S_j are distinct if $i \neq j$.

It becomes nontrivial to find $\mathsf{V}(\Sigma, D)$ when D is fragmented and distributed. As shown in Example 1, to detect the violations in distributed D_0, it is necessary to ship data from one site to another. Hence a natural question concerns how to find $\mathsf{V}(\Sigma, D)$ with minimum amount of data shipment. That is, we want to reduce communication cost and network traffic.

To characterize the communication cost, we use $M(i, j)$ to denote the set of tuples shipped from S_i to S_j, and M the total data shipment, *i.e.,* $\bigcup_{i,j \in [1,n], i \neq j} M(i, j)$.

For each $j \in [1, n]$, we use $D_j(M)$ to denote fragment D_j augmented by data shipped in M, *i.e.,* $D_j(M)$ includes data in D_j and all the tuples in M that are shipped to site S_j. More specifically, for vertical partitions,

$$D_j(M) = D_j \bowtie_{i \in [1,n] \wedge M(i,j) \neq \emptyset} M(i, j);$$

while for horizontal partitions,

$$D_j(M) = D_j \cup \bigcup_{i \in [1,n] \wedge M(i,j) \neq \emptyset} M(i, j).$$

We say that a CFD ϕ can be *checked locally after data shipments* M if $\mathsf{V}(\phi, D) = \bigcup_{i \in [1,n]} \mathsf{V}(\phi, D_i(M))$. As a special case, we say that ϕ can be *checked locally* if $\mathsf{V}(\phi, D) = \bigcup_{i \in [1,n]} \mathsf{V}(\phi, D_i)$, *i.e.,* all violations of ϕ in D can be found at individual site without data shipment (*i.e.,* $M = \emptyset$).

A set Σ of CFDs *can be checked locally after* M if each ϕ in Σ can be checked locally after M.

The *distributed CFD detection problem with minimum communication cost* is to determine, given a positive number K, a set Σ of CFDs and a partitioned and distributed relation D, whether there exists a set M of data shipments such that (1) Σ can be checked locally after M, and (2) the size $|M|$ of M is no larger than K, *i.e.,* $|M| \leq K$.

In contrast to the error detection problem in centralized data, it is beyond reach in practice to find an efficient algorithm to detect errors in distributed data with minimum network traffic [10].

Theorem 1 [10]: *The distributed CFD detection problem with minimum communication cost is NP-complete, when data is either vertically or horizontally partitioned.* □

In light of the intractability, a heuristic algorithms was developed in [10] to compute $\mathsf{V}(\Sigma, D)$ when D is horizontally partitioned. We are not aware of any algorithm for detecting CFD violations for data that is vertically partitioned.

3 Incremental Detection: Complexity

We formulate the incremental detection problem and study its complexity. We start with notations for updates.

Updates. We consider a *batch update* ΔD to a database D, which is a list of tuple insertions and deletions. A modification is treated as an insertion after a deletion. We use ΔD^+ to denote the sub-list of all tuple insertions in ΔD, and ΔD^- the sub-list of deletions in ΔD. We use $D \oplus \Delta D$ to denote the updated database of D with ΔD.

In a vertical partition $D = (D_1, \ldots, D_n)$ (see Section 2), we write $\Delta D_i = \pi_{X_i}(\Delta D)$ for updates in ΔD to fragment D_i. For a horizontal partition, we denote the updates to D_i as $\Delta D_i = \sigma_{F_i}(\Delta D)$; similarly for ΔD_i^+ and ΔD_i^-.

Problem statement. Given D, ΔD and a set Σ of CFDs, we want to find $\mathsf{V}(\Sigma, D \oplus \Delta D)$, *i.e.,* all violations of CFDs of Σ in the updated database $D \oplus \Delta D$.

As remarked earlier, we want to minimize unnecessary recomputation by *incrementally* computing $\mathsf{V}(\Sigma, D \oplus \Delta D)$. More specifically, suppose that the old output $\mathsf{V}(\Sigma, D)$ is also provided. *Incremental detection* is to find the *changes* $\Delta\mathsf{V}$ to $\mathsf{V}(\Sigma, D)$ such that $\mathsf{V}(\Sigma, D \oplus \Delta D) = \mathsf{V}(\Sigma, D) \oplus \Delta\mathsf{V}$. We refer to this as the *incremental detection problem*.

In practice, when ΔD is small, $\Delta\mathsf{V}$ is often small as well. Hence it is more efficient to find $\Delta\mathsf{V}$ rather than *batch detection* that recomputes $\mathsf{V}(\Sigma, D \oplus \Delta D)$ starting from scratch. That is, we maximally reuse the old output $\mathsf{V}(\Sigma, D)$ when computing the new output $\mathsf{V}(\Sigma, D \oplus \Delta D)$.

We use $\Delta\mathsf{V}^+$ to denote $\mathsf{V}(\Sigma, D \oplus \Delta D) \setminus \mathsf{V}(\Sigma, D)$, *i.e.,* violations added, and $\Delta\mathsf{V}^-$ for $\mathsf{V}(\Sigma, D) \setminus \mathsf{V}(\Sigma, D \oplus \Delta D)$, *i.e.,* violations removed. Then $\Delta\mathsf{V} = \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}^-$. Observe that ΔD^+ only incurs $\Delta\mathsf{V}^+$, and ΔD^- only leads to $\Delta\mathsf{V}^-$.

When D is partitioned into $(D_1, \ldots, D_n)$ and distributed, we say that $\Delta\mathsf{V}$ can be *computed locally* after data shipments M of tuples from $D \oplus \Delta D$ if $\Delta\mathsf{V} = \bigcup_{i \in [1,n]} \Delta\mathsf{V}_i(M)$, where $\Delta\mathsf{V}_i(M)$ denotes the differences between $V(\Sigma, D_i(M) \oplus \Delta D_i)$ and $V(\Sigma, D_i)$ at site S_i.

The *incremental* distributed CFD detection problem with minimum communication cost is to find, given D, Σ, $\Delta D, \mathsf{V}(\Sigma, D)$ as input, $\Delta\mathsf{V}$ with *minimum* data shipments M such that $\Delta\mathsf{V}$ is locally computable after M.

Its decision problem is to determine, given D, Σ, $\Delta D, \mathsf{V}(\Sigma, D)$ and a positive number K, whether there exists a set M of data shipments such that (1) $\Delta\mathsf{V}$ can be computed locally after M, and (2) $|M| \leq K$. We refer to the problem as IMVD for vertically partitioned data, and as IMHD for horizontally partitioned data.

In practice, the set Σ of CFDs is typically predefined and is rarely changed, although D is frequently updated. Thus in the sequel we consider fixed Σ.

Intractability results. Unfortunately, incremental detection is no easier than its batch counterpart (Theorem 1).

Below we shall first study the case for vertical partitions, then analyze its horizontal counterpart.

Theorem 2: *The incremental distributed CFD detection problem with minimum data shipment is NP-complete for vertical partitions (IMVD). It remains NP-hard for fixed CFDs when (a) update consists of insertions only, for a fixed database with fixed partitions, or (b) update consists of deletions only.* □

Proof. Upper bound. To show that IMVD is in NP, we provide an NP algorithm for incremental detection of violations in vertical partitions. It works as follows: first guess a set M of data shipments such that $|M| \leq K$, and then inspect whether $\Delta\mathsf{V} = \bigcup_{i\in[1,n]} \Delta\mathsf{V}_i(M)$. The checking can be done in PTIME.

Lower bound. We show that IMVD is NP-hard even when (1) ΔD consists of insertions only, or (2) ΔD consists of deletions only. We use fixed CFDs in both cases.

(1) When ΔD consists of insertions only. We verify the NP-hardness of IMVD by reduction from the minimum vertical detection problem (MVD). Given a set Σ of CFDs, a vertically partitioned database D and a positive number K, MVD is to decide whether there exists a set M of data shipments such that Σ can be checked locally after M, and $|M| \leq K$. It is known that MVD is NP-complete for a *fixed set* Σ defined on a fixed schema [10].

Given an instance (Σ, D, K) of MVD, we construct an instance $(\Sigma, D', \mathsf{V}(\Sigma, D), \Delta D^+, K)$ of IMVD by letting $D' = \emptyset$, $\Delta D^+ = D'$ and $\mathsf{V}(\Sigma, D) = \emptyset$. One can verify that there is M such that $|M| \leq K$ and Σ can be checked locally after M iff there exists a set M' of data shipments such that $|M'| \leq K$ and $\Delta\mathsf{V}$ can be computed locally after M'. Note that $D' = \emptyset$ is independent of input (Σ, D, K). In other words, IMVD is NP-hard when the CFDs, the database and its partition are all fixed.

(2) When ΔD consists of deletions only. We show the NP-hardness of IMVD also by reduction from MVD. Given an instance (Σ, D, K) of MVD, we define an IMVD instance as follows. Assume that Σ is defined on schema R.

(a) We define a new schema $R' = R \cup \{B_1, B_2\}$, where B_1 and B_2 are distinct attributes not appearing in R.

(b) We define the set of $\Sigma' = \Sigma \cup \{\varphi\}$, where φ is an FD $B_1 \to B_2$. Assume *w.l.o.g.* that there exist two distinct values v_1 and v_2 in the domains of B_1 and B_2.

(c) We define D' such that for each $t_i \in D$, D' includes two tuples t_{ai} and t_{bi}, where $t_{ai}[R] = t_{bi}[R] = t[R]$, $t_{ai}[B_1B_2] = (v_1, v_1)$, and $t_{bi}[B_1B_2] = (v_1, v_2)$. That is, if D consists of n tuples. D' consists of $2 * n$ tuples. The relations D and D' have the same partitions for all the attributes in R. In addition, a new fragment of D' is added, consisting of new attributes B_1, B_2 and the key attribute key of D. Obviously, $\mathsf{V}(\Sigma, D') = D'$, since every tuple of D' violates φ with another tuple in D'.

(d) We define the set ΔD^- of *deletions* to be $\{t_{bi} \mid i \in [1, n]\}$, *i.e.*, it is to remove all tuples t_{bi}.

To see that these make a reduction, observe the following. Before D is updated by ΔD^-, $\mathsf{V}(\Sigma, D') = D'$. After D is updated, $\mathsf{V}(\Sigma, D' \oplus \Delta D^-) = \mathsf{V}(\Sigma, D)$. From this it follows that a solution (a set of data shipments) to (Σ, D, K) iff it is a solution to $(\Sigma, D', \mathsf{V}(\Sigma, D), \Delta D^-, K)$. Moreover, since MVD is NP-complete when Σ and fragmentation are fixed, so is IMVD when ΔD consists of deletions only, since the newly added φ and the refined fragmentation are also independent of the input. □

We next analyze the case for horizontal partitions.

Theorem 3: *The incremental distributed CFD detection problem with minimum data shipment is NP-complete for horizontally partitioned data (IMHD). It remains NP-hard for fixed CFDs and for (a) insertions only, with a fixed database with fixed partitions, or (b) for deletions only.* □

Proof. Upper bound. We show that IMHD is in NP by providing an NP algorithm for IMHD. It works as follows: first guess a set M of data shipments such that $|M| \leq K$, and then inspect whether $\Delta\mathsf{V} = \bigcup_{i\in[1,n]} \Delta\mathsf{V}_i(M)$. The latter can be done in PTIME.

Lower bound. We show that IMHD is NP-hard for fixed CFDs even when (1) ΔD consists of insertions only with a fixed D, or (2) ΔD consists of deletions only.

(1) When ΔD consists of insertions only. We show that IMHD is NP-hard by reduction from the minimum set cover problem (MSC). Given a finite set X of elements, a collection $\mathcal{C}$ of subsets of X and a positive number K, MSC is to decide whether there exists a cover for X of size K or less, *i.e.*, a subset $C' \subseteq \mathcal{C}$ such that $|C'| \leq K$ and every element of X belongs to at least one member of C'. It is known that MSC is NP-complete even when each subset in $\mathcal{C}$ has three elements (cf. [13]).

Given an instance $(X, \mathcal{C}, K)$ of MSC, we construct an instance (Σ, D, $\mathsf{V}(\Sigma, D)$, ΔD^+, K') of IMHD such that the IMHD problem has a solution iff the MSC problem has a solution. Assume *w.l.o.g.* that $X = \{x_j \mid j \in [1, m]\}$, $\mathcal{C} = \{C_i \mid i \in [1, n]\}$, each C_i consists of three elements of X, and that $X = \bigcup_{i\in[1,n]} C_i$ (*i.e.*, there exists a cover).

(a) We define schema $R = (A_1, A_2, A_3, B, N, L)$. Intuitively, A_1, A_2, A_3 are to encode the three elements in a subset C_i of $\mathcal{C}$, B for type (*i.e.*, a subset or an element), N is a partition key, and L is a tuple id within the fragment.

(b) The set Σ consists of three fixed FDs: $A_i \to B$, $i \in [1, 3]$.

(c) We construct an instance D of R that is horizontally partitioned into 2 fragments D_u and D_v, residing at sites S_u and S_v, respectively. Assume an arbitrary topological order $\prec$ on the elements of X, and four fixed distinct values b_1, b_2, u and v. Tuples in D are partitioned into D_u and D_v with the selection predicate as $N = u$ and $N = v$, respectively. Initially, D is empty, and hence, both D_u and D_v are empty. Thus so are $\mathsf{V}(\Sigma, D_u)$ and $\mathsf{V}(\Sigma, D_v)$.

(d) We define *insertions* ΔD^+ as follows.

- ΔD^+_u consists of $(n + m)$ tuples. For each $i \in [1, n]$, there exists a tuple t_{c_i} in ΔD^+_u such that $t_{c_i} = (a_1, a_2, a_3, b_1, u, i)$, where a_1, a_2, a_3 are the elements in

$C_i \in \mathcal{C}$, such sorted that $a_1 \prec a_2 \prec a_3$. For each i in $[1, m]$, there exists a tuple t_{x_i} in ΔD_u^+, such that $t_{x_i} = (x_i, x_i, x_i, b_2, u, i+n)$. Intuitively, each t_{c_i} encodes a subset C_i, and each t_{x_i} encodes an element of X.
- ΔD_v^+ consists of $m*(n+1)$ tuples. For each $i \in [1, m]$, there exist $(n+1)$ tuples $t_{x_i 1}$, $t_{x_i 2}$ $\ldots$, $t_{x_i(n+1)}$ in ΔD_v^+, such that $t_{x_i j} = (x_i, x_i, x_i, b_2, v, (i-1)*(n+1)+j)$, for $j \in [1, n+1]$. Intuitively, for each $i \in [1, m]$, there exist $(n+1)$ tuples that encode x_i.

Assume *w.l.o.g.* that tuples in ΔD^+ have the same size l.

(e) We define K' to be $K * l$.

Observe that schema R, database D and CFDs Σ are all *fixed*, *i.e.,* they are independent of the MSC instance.

Intuitively, for all tuples $t \in \Delta D^+$, if $t[B] = b_1$, then t encodes a subset $C_i \in \mathcal{C}$; and if $t[B] = b_2$, then t encodes an element x_i in X. In addition, t_1 and t_2 in ΔD^+ violate a CFD of Σ if one of them is a tuple encoding a subset C_i, the other encodes an element x_i, and $x_i \in C_i$. All the tuples in ΔD_u^+ and ΔD_v^+ violate some CFDs of Σ. Note that only violations incurred by tuples t_{x_i} and t_{c_j} in ΔD_u^+ can be detected locally, without requiring data shipment. Tuples in ΔD_v^+ do not cause local violations; but for each tuple $t_{x_i j}$ there exists a tuple t_{c_k} in ΔD_u^+ such that $t_{x_i j}$ and t_{c_k} violate a CFD, where x_i is an element of C_k, $i \in [1, m], j \in [1, n+1]$, and $k \in [1, n]$. Intuitively, to detect violations in ΔD_v^+ locally, a *"cover"* $C' \subseteq \mathcal{C}$ of X must be shipped from site S_u to S_v.

We now show that $(\Sigma, D, \mathsf{V}(\Sigma, D), \Delta D^+, K')$ is indeed a reduction from MSC to IMHD. First, assume that the MSC instance has a cover C' of size no larger than K. We define a set M of tuple shipments $M = \{t_{c_i} \mid C_i \in C'\}$. We ship M from site S_u to S_v. Note that the size of M is no larger than K'. Since C' is a cover, at site S_v, all tuples $t \in D_v(M) \oplus \Delta D_v^+$ can be detected as violations locally. Hence, $\Delta\mathsf{V}_u(M) \cup \Delta\mathsf{V}_v(M) = \Delta\mathsf{V}_u \cup \Delta\mathsf{V}_v(M) = \Delta D_u^+ \cup \Delta D_v^+ \cup M = \Delta D_u^+ \cup \Delta D_v^+ = \Delta\mathsf{V}$.

Conversely, assume that there exists a set M of tuple shipments such that $|M| \leq K' = K * l$, and after M, $\Delta\mathsf{V}$ can be computed locally. (a) If $K' = n * l$, then the set $\mathcal{C}$ consisting of all subsets is a cover and $|\mathcal{C}| \leq n \leq K$. (b) When $K' < n * l$, let $M = M_{u \to v} \cup M_{v \to u}$, where $M_{u \to v}$ (resp. $M_{v \to u}$) denotes the part of M shipped from S_u (resp. S_v) to S_v (resp. S_u). Since $|M_{v \to u}| \leq |M| \leq K'$, there are no more than n tuples in $M_{v \to u}$. Thus for any element $x_i \in X$, there exists at least one tuple $t_{x_i j} \in \Delta D_v^+ \setminus M_{v \to u}$. Since each $t_{x_i j}$ is detected as a local violation, each x_i has to be covered by tuple t_{c_k} in $M_{u \to v}$, which encodes a subset C_k. Let $C' = \{C_k \mid t_{c_k} \in M_{u \to v}\}$. Then C' is indeed a cover of X, and $|C'| \leq K$.

(2) When ΔD consists of deletions only. We show that IMHD is NP-hard also by reduction from MSC.

Given an instance (X, C, K) of MSC, we construct an instance $(\Sigma, D', \mathsf{V}(\Sigma, D'), \Delta D^-, K')$ such that the IMHD problem has a solution iff MSC has a solution.

We use the same R, Σ and K' as defined in (1) above. An instance D' is also partitioned into D'_u and D'_v with the same predicates given in (1). More specifically,
- $D'_u = \Delta D_u^+$, consisting of $(n+m)$ tuples given in (1);
- D'_v consists of $(m*(n+1)+n)$ tuples, in which $m*(n+1)$ tuples are from ΔD_v^+ given in (1). The other n tuples are given as follows. For each $i \in [1, n]$, D'_v includes a tuple $t'_{c_i} = (a_1, a_2, a_3, b_1, v, m*(n+1)+i)$, where a_1, a_2, a_3 are the elements in $C_i \in \mathcal{C}$, such sorted that $a_1 \prec a_2 \prec a_3$ for some order $\prec$.

We define *deletions* ΔD^- to be $\{t'_{c_i} \mid i \in [1, n]\}$, *i.e.,* it is to remove all those tuples t'_{c_i} from D'_v. Here $\mathsf{V}(\Sigma, D') = D'$, *i.e.,* every tuple in D' is a violation of some CFD in Σ.

Note that schema R and CFDs Σ are both fixed, *i.e.,* they are independent of the MSC instance.

Observe that before D' is updated by ΔD^-, all the violations can be detected locally in D'_u and D'_v. After D' is updated, $D' \oplus \Delta D^-$ became the relation D given in (1) above, and $\mathsf{V}(\Sigma, D' \oplus \Delta D^-)$ = $\mathsf{V}(\Sigma, D)$. Hence along the same lines as the proof for (1), one can verify that $(\Sigma, D', \mathsf{V}(\Sigma, D'), \Delta D^-, K')$ is a reduction from MSC. □

From the proofs of Theorem 2 and 3, it follows:

Corollary 4: *The incremental distributed CFD detection problems* IMVD *and* IMHD *with minimum data shipment remains NP-complete even for fixed FDs only.* □

The boundedness result. Not all is lost. As observed in [27], the cost of an *incremental algorithm* should be analyzed in terms of the size of the *changes* in both input and output, denoted as $|\Delta\mathsf{C}|$, rather than the size of the entire input. Indeed, $|\Delta\mathsf{C}|$ characterizes the updating costs *inherent to* the incremental problem itself.

An incremental problem is said to be *bounded* if its cost can be expressed as a function of $|\Delta\mathsf{C}|$. An incremental algorithm is *optimal* if its cost is in $O(|\Delta\mathsf{C}|)$; *i.e.,* it only does the amount of work that is *necessary* to be performed by any incremental algorithm for the problem. In other words, it is the best one can hope for.

For incremental violation detection, $|\Delta\mathsf{C}| = |\Delta D| + |\Delta\mathsf{V}|$. It is *bounded* if its communication and computational costs are both functions of $|\Delta\mathsf{C}|$, *independent of* $|D|$.

Although the distributed incremental detection problem is NP-complete *w.r.t.* minimum data shipment (Theorems 2 and 3), the good news is that it is bounded *w.r.t.* the changes in both input and output.

Theorem 5: *The incremental distributed CFD detection problem is bounded for data partitioned vertically or horizontally. There are optimal incremental detection algorithms with communication and computational costs in $O(|\Delta\mathsf{C}|)$.* □

In the rest of the paper, we prove Theorem 5 by providing optimal algorithms for data that is partitioned vertically (Section 4) or horizontally (Sections 6).

4 Algorithms for Vertical Partitions

We start with an *optimal* incremental detection algorithm for vertical partitions $D = (D_1, \ldots, D_n)$. Here for

$i \in [1, n]$, D_i resides at site S_i and $D_i = \pi_{X_i}(D)$ (see Section 2). The main result of this section is as follows.

Proposition 6: *There exists an algorithm that incrementally detects* CFD *violations in vertical partitions with communication and computational costs in* $O(|\Delta D| + |\Delta \mathsf{V}|)$. $\square$

It is nontrivial to develop an incremental detection algorithm bounded by $O(|\Delta D| + |\Delta \mathsf{V}|)$. To find $\Delta\mathsf{V}$, not only tuples in ΔD but also data in D may be needed and hence shipped. Indeed, as in Example 2, to validate ϕ_1 after t_6 is inserted into D_0 of Fig. 1, $t_5[\mathsf{street}, \mathsf{city}]$ in D_{V_2} and $t_5[\mathsf{CC}]$ in D_{V_3} are necessarily involved.

Below we shall first identify when the data in D is not needed in incremental detection. For the cases when the involvement of D is inevitable, we propose index structures to avoid shipping data in D. Based on the auxiliary structures, we then develop an optimal algorithm for vertically partitioned databases.

Cases independent of D. To validate a CFD $\phi = (X \to B, t_p)$ in response to the insertion or deletion of a tuple t, data in D is not needed in the following two cases.

(1) When ϕ is a *constant* CFD. Indeed, ϕ can be violated by a single tuple t alone. Hence to find $\Delta\mathsf{V}$ incurred by t, there is no need to consult other tuples in D.

(2) When ϕ is a variable CFD with $X \cup \{B\} \subseteq X_i$. In this case, ϕ can be *locally checked* at site S_i in which $D_i = \pi_{X_i}(D)$ resides. There is no need to ship data.

Index structures. Below we focus on validation of variable CFD $\phi = (X \to B, t_p)$, *i.e.*, $t_p[B] =$ '_'.

Observe that for a tuple t to make a violation of a CFD ϕ, there must exist some tuple t' such that $t[X] = t'[X]$, and moreover, either (a) $t[B] = t'[B]$ and t is already a violation of the CFD ϕ, or (b) $t[B] \neq t'[B]$, *i.e.*, $(t, t') \not\models \phi$. To capture this, we define an equivalence relation *w.r.t.* a set Y of attributes.

Equivalence classes. We say that tuples t and t' are *equivalent w.r.t.* Y if $t[Y] = t'[Y]$. We denote by $[t]_Y$ the equivalence class of t, *i.e.*, $[t]_Y = \{t' \in D \mid t'[Y] = t[Y]\}$. We associate a unique identifier (eqid) $\mathsf{id}[t_Y]$ with $[t]_Y$.

We define a function $\mathsf{eq}()$ that takes as input the eqid's of equivalence classes $[t]_{Y_i}$ ($i \in [1, m]$), and returns the eqid of $[t]_Y$, where $Y = \bigcup_{i\in[1,m]} Y_m$, *i.e.*, $\mathsf{eq}(\mathsf{id}[t_{Y_1}], \cdots, \mathsf{id}[t_{Y_m}]) = \mathsf{id}[t_Y]$. As will be seen shortly, we send $\mathsf{id}[t_Y]$ rather than data in $[t]_{Y_i}$ to reduce the amount of data shipped.

Upon $[t]_Y$'s, we define the following index structures.

HEV-*index*. For each variable CFD $\phi = (X \to B, t_p)$, each sites S_i maintains a set of **H**ash-based **E**quivalence class and **V**alue indices (HEV's), denoted by HEV_i^{ϕ}. Each non-base HEV is a *key/value* store that given a tuple t and a set of eqid's $\mathsf{id}[t_{Y_j}]$ ($j \in [1, m]$) as the *key*, returns $\mathsf{id}[t_{Y_1 \cup \cdots \cup Y_m}]$ as the *value*. *Base* HEV's are also maintained to map distinct attribute values to their eqid's. These are special HEV's that take single attribute values as the key, and are shared by all CFDs. We write HEV_i for HEV_i^{ϕ} when ϕ is clear from the context.

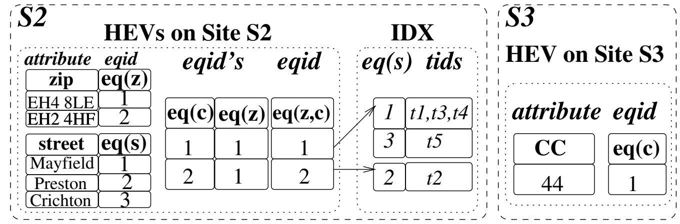


Fig. 3. Example HEV-indices and an IDX for ϕ_1

Intuitively, HEV's help us identify $\mathsf{id}[t_X]$ and $\mathsf{id}[t_B]$, since all tuples that violate ϕ with t must be in $[t]_X$, and on attribute B, they have different values from $t[B]$.

The HEV's for CFD ϕ are organized as follows. We build HEV_X and HEV_B for attributes X and B, respectively. More specifically, we sort attributes of X into $(x_1, \ldots, x_m)$, and for each $i \in [1, m]$, we build an HEV for the subset $\{x_j \mid j \in [1, i]\}$. As will be seen in Example 5, to identify $\mathsf{id}[t_X]$, we use the HEV's for $\{x_1\}$, $\{x_1, x_2\}$, ..., $\{x_1, \ldots, x_m\}$ one by one in this order. We shall present the details of the strategy for building HEV's in Section 5, which aims to reduce eqid shipment when multiple CFDs are taken together.

IDX. We group tuples that violate ϕ with t into $[t']_{X\cup\{B\}}$ for each t' in $[t]_X$. The tuples are indexed by IDX, another hash index that is only stored at the site where $\mathsf{id}[t_X]$ is maintained. Given a tuple t, it returns a $\mathsf{set}(t[X])$ of distinct eqid's of $[t']_{X\cup\{B\}}$, where $t[X] = t'[X]$, and each eqid in turn identifies the set of all tuple ids in the equivalence class $[t']_{X\cup\{B\}}$. Intuitively, for each $[t]_X$, an IDX stores distinct values of B attribute and their associated tuple ids.

Example 5: Figure 3 depicts HEV's for ϕ_1 of Fig. 1 and relation D_0 of Fig. 2. HEV_2 and HEV_3 are the indices on sites S_2 and S_3, respectively, and the IDX is stored at S_2.

To compute $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$, we first find $\mathsf{id}[t_{5\{\mathsf{CC}\}}] = 1$ from a base hash table of HEV_3, since $t_5[\mathsf{CC}] = 44$, at site S_3. The eqid 1 (*i.e.*, $\mathsf{id}[t_{5\{\mathsf{CC}\}}]$) is then sent to S_2. Using the base hash table at site S_2, we get $\mathsf{id}[t_{5\{\mathsf{zip}\}}] = 1$ from $t_5[\mathsf{zip}] =$ EH4 8LE. Taking these together as the input for HEV_2, we get $\mathsf{eq}(1, 1) = 1$, which is for $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$.

Moreover, as shown in Fig. 3, $\mathsf{id}[t_{5\{\mathsf{CC},\mathsf{zip}\}}]$ links to two entries in IDX, where 1 represents Mayfield with an equivalence class $\{t_1, t_3, t_4\}$, and 3 indicates Crichton with an equivalence class $\{t_5\}$.

Observe that during the detection, we use HEV's for eqid's of any tuple in this order: $\{\mathsf{CC}\}$ and $\{\mathsf{CC}, \mathsf{zip}\}$. $\square$

Example 5 tells us that to identify $\mathsf{id}[t_X]$, one only needs to ship at most $|X| - 1$ eqid's, to make the input for HEV_X, *i.e.*, the index of X.

Algorithms. Leveraging the index structures, we develop an incremental algorithm to detect violations in vertical partitions. To simplify the discussion, we first consider a single update for a single CFD. We then extend the algorithm to multiple CFDs and batch updates.

Single update for one CFD. Given a CFD ϕ, a vertically partitioned database D, violations $\mathsf{V}(\phi, D)$ of Σ in D, and a tuple t inserted into (resp. deleted from) D, the algo-

Algorithm incVIns
Input: ΔD^+=$\{t\}$, a vertically partitioned D, a variable CFD ϕ and the old violations $\mathsf{V}(\phi, D)$.
Output: $\Delta\mathsf{V}^+$.

/* $\phi = (X \rightarrow B, t_p)$ */
1. identify $\mathsf{set}(t[X])$ using HEV's and IDX's;
2. **if** $|\mathsf{set}(t[X])| > 1$ **then** $\Delta\mathsf{V}^+ := \{t\}$;
3. **elseif** $|\mathsf{set}(t[X])| = 1$ (*i.e.*, $\mathsf{set}(t[X]) = \{t'\}$) **then**
4. **if** $(t, t') \not\models \phi$ **then** $\Delta\mathsf{V}^+ := \{t\} \cup [t']_{X \cup \{B\}}$;
5. **else** $\Delta\mathsf{V}^+ := \emptyset$;
6. **else** $\Delta\mathsf{V}^+ := \emptyset$;
7. augment IDX by adding t; HEV-indices are also maintained;
8. **return** $\Delta\mathsf{V}^+$;

Algorithm incVDel
Input: ΔD^-=$\{t\}$, a vertical partition D, a variable CFD ϕ and $\mathsf{V}(\phi, D)$.
Output: $\Delta\mathsf{V}^-$.

/* $\phi = (X \rightarrow B, t_p)$ */
1. identify $\mathsf{set}(t[X])$ and $[t]_{X \cup \{B\}}$ using HEV's and IDX's;
2. **if** $|[t']_{X \cup \{B\}}| > 1$
3. **if** $|\mathsf{set}(t[X])| > 1$ **then** $\Delta\mathsf{V}^- := \{t\}$;
4. **else** $\Delta\mathsf{V}^- := \emptyset$;
5. **else** /* $|[t']_{X \cup \{B\}}| = 1$ */
6. **if** $|\mathsf{set}(t[X])| > 2$ **then** $\Delta\mathsf{V}^- := \{t\}$;
7. **elseif** $|\mathsf{set}(t[X])|$=2 (*i.e.*, $\{t, t'\}$) **then** $\Delta\mathsf{V}^-$:=$\{t\} \cup [t']_{X \cup \{B\}}$;
8. **else** $\Delta\mathsf{V}^- := \emptyset$;
9. maintain IDX by deleting t; HEV-indices are also maintained;
10. **return** $\Delta\mathsf{V}^-$;

Fig. 4. Single Insertion/Deletion for Vertical Partitions

rithm identifies changes $\Delta\mathsf{V}^+(\phi, D)$ (resp. $\Delta\mathsf{V}^-(\phi, D)$) to $\mathsf{V}(\phi, D)$. It first uses HEV to find the equivalence classes $[t]_X$ and its associate sets in IDX. It then computes $\Delta\mathsf{V}$.

<u>*Insertions*</u>. The algorithm for single-tuple insertion is shown in Fig. 4, referred to as incVIns. It first identifies $\mathsf{set}(t[X])$ by capitalizing on HEV-indices as discussed above (line 1). This requires to ship at most X eqid's, including the eqid of $t[B]$. When $|\mathsf{set}(t[X])| > 1$, all tuples t' such that (t', t) violate ϕ must have been found. Hence t is the only new violation (line 2; see Example 2). When $|\mathsf{set}(t[X])| = 1$, there are two cases: (1) if $\mathsf{set}(t[X])$ contains the entry for tuple t', where (t, t') violate ϕ, then t and all tuples in $[t']_{X \cup \{B\}}$ are new violations (line 4); and (2) if $\mathsf{set}(t[X])$ only contains the entry for t, then no violation arises (line 5). Otherwise, no tuple agrees with t on X attributes, and there is no violation (line 6). The new violations in $\Delta\mathsf{V}^+$ are then returned (line 8).

The index IDX is maintained in the same process, by inserting a tuple t into the set $[t]_{X \cup \{B\}}$, or adding an new entry to $\mathsf{set}(t[X])$ and its associated set $[t]_{X \cup \{B\}} = \{t\}$. In either case, it takes constant time. The HEV-indices are updated together with $\mathsf{id}[t_X]$. If such an eqid does not exist, a new entry is generated and added to the corresponding HEV-indices (line 7).

<u>*Deletions*</u>. The algorithm for single-tuple deletions, denoted as incVDel, is also shown in Fig. 4. It first finds both $[t]_{X \cup \{B\}}$ and $\mathsf{set}(t[X])$ using HEV (line 1). If no tuples are in $[t]_{X \cup \{B\}}$ after t is deleted (line 2), t is the only violation removed (line 3); otherwise there is no change to $V(\phi, D)$ (line 4). If t is the only tuple in $[t]_{X \cup \{B\}}$ (line 5), *i.e.*, the entry of t in $\mathsf{set}(t[X])$ will be removed, there are three cases to consider: (1) all violations *w.r.t.* t remain, and only t is removed (line 6); (2) all violations *w.r.t.* t are

Algorithm incVer
Input: ΔD, D in n vertical partitions, a set Σ of CFDs and $\mathsf{V}(\Sigma, D)$.
Output: $\Delta\mathsf{V}$.
1. remove updates in ΔD with the same tuple id and canceling each other;
2. $\Delta\mathsf{V}^- := \emptyset$; $\Delta\mathsf{V}^+ := \emptyset$;
3. **for each** $\phi \in \Sigma$ **do**
4. **if** ϕ is a constant CFD **then** /* $\phi = (X \rightarrow B, t_p)$ */
5. $T_i := \{t \mid t \in \Delta D$ and $t[X_i \cap X] \asymp t_p[X_i \cap X]\}$ for $i \in [1, n]$;
6. ship all T_i with their values on B attribute to one site;
7. merge T_i for $i \in [1, n]$ based on the same tuple id, get T;
8. **for each** $t \in T$ **do**
9. **if** $t[B] = t_p[B]$ **and** $t \in \Delta D^-$ **then** $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \{t\}$;
10. **elseif** $t[B] \neq t_p[B]$ **and** $t \in \Delta D^+$ **then** $\Delta\mathsf{V}^+$:=$\Delta\mathsf{V}^+ \cup \{t\}$;
11. **elseif** ϕ can be locally checked at S_i **then**
12. derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ at S_i use HEV_i and IDX (Section 4);
13. $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$; $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$
14. **else** /* a variable CFD that cannot be *locally checked* */
15. derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ ($i \in [1, n]$) (see Fig. 4);
16. $\Delta\mathsf{V}^-$:=$\Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$ **and** $\Delta\mathsf{V}^+$:=$\Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$ ($i \in [1, n]$);
17. **return** $\Delta\mathsf{V} = \Delta\mathsf{V}^- \cup \Delta\mathsf{V}^+$;

Fig. 5. Batch Updates for Vertical Partitions

removed together with t when t is deleted (line 7); or (3) t does not violate ϕ (line 8). HEV and IDX indices are maintained similar to the case for insertions (line 9). Finally, $\Delta\mathsf{V}^-$ is returned (line 10).

Example 6: Consider D_0 (without t_6) of Fig. 2, ϕ_1 of Fig. 1, and its indices given in Fig. 3. When t_6 is inserted, at site S_3, it identifies $\mathsf{eq}(\mathsf{id}[t_{6\{\mathsf{CC}\}}]) = 1$ ($t_6[\mathsf{CC}] = 44$) from HEV_3 and ships this eqid (*i.e.*, 1) to S_2. At S_2, it identifies $\mathsf{eq}(\mathsf{id}[t_{6\{\mathsf{zip}\}}]) = 1$ ($t_6[\mathsf{CC}]$ = EH8 4LE) and $\mathsf{eq}(1, 1) = 1$. This links to two entries in IDX as shown in Fig. 3, indicating that t_6 is the only new violation, *i.e.*, $\Delta\mathsf{V}^+ = \{t_6\}$ (line 2). Indeed, $\{t_5, t_6\} \not\models \phi_1$ and t_5 is a known violation. Only a single eqid (*i.e.*, 1) is shipped from site S_3 to site S_2.

Now suppose that tuple t_4 is deleted. Algorithm incVDel will find the eqid of $[t_4]_{\{\mathsf{CC},\mathsf{zip}\}}$ to be 1, which links to two entries, following the same process as above. After t_4 is deleted, $[t_4]_{\{\mathsf{CC},\mathsf{zip}\}}$ is not empty, *i.e.*, $[t_4]_{\{\mathsf{CC},\mathsf{zip}\}} = \{t_1, t_3\}$. Hence $\Delta\mathsf{V}^- = \{t_4\}$ (line 3). Again only a single eqid (*i.e.*, 1) is shipped. □

<u>***Batch updates and multiple CFDs***</u>. We now present an algorithm, denoted as incVer in Fig. 5, that takes *batch updates* ΔD, a vertically partitioned D, a set Σ of CFDs, and violations $\mathsf{V}(\Sigma, D)$ of Σ in D as input. It finds and returns the changes $\Delta\mathsf{V}$ of violations to $\mathsf{V}(\Sigma, D)$.

The algorithm works as follows. It first removes the updates in ΔD that cancel each other (line 1), and initializes the changes (line 2). It then detects the changes of violations for multiple CFDs in parallel (lines 3-16). It deals with three cases. (1) *Constant* CFDs (lines 4-10). It first identifies at each site S_i the tuple ids that can possibly match the pattern tuple t_p (line 5). These identified (partial) tuples are shipped to a designated coordinator site, together with corresponding B values (line 6). These tuple ids are naturally sorted in ascending order (by indices). A sort merge of them is thus conducted in linear time, and it generates a set T of tuples in which each tuple matches the pattern tuple t_p on X attributes (line 7). It then examines these tuples' B attributes, to decide whether they are violations to be removed (line 9), or violations newly incurred (line 10). (2) *Locally checked variable* CFDs (lines 11-13). The changes of violations can be detected using the same indices as for a single CFD given above (lines 12-13). (3) *General variable* CFDs (lines 14-16). The method used is exactly what we have seen for a single CFD. The changes to violations are then returned (line 17).

Violations are marked with those CFDs that they violate when combining ΔV's for multiple CFDs (see Fig. 1).

Complexity. For the communication cost, note that only eqid's are sent: for each tuple $t \in \Delta D$ and each CFD $\phi \in \Sigma$, its eqid's are sent at most $|X|$ times. As remarked earlier, the set Σ of CFDs and the fragmentation are fixed as commonly found in incremental integrity checking. Hence the messages sent are bounded by $O(|\Delta D|)$. The computational cost is in $O(|\Delta D| + |\Delta \mathsf{V}|)$, since checking both hash-based HEV and IDX take constant time, as well as their maintenance for each update.

5 Optimization for Vertical Partitions

We have seen that by leveraging HEV's and IDX's, for vertical partition an incremental detection algorithm can be developed that is *bounded* in the changes in the input and output (*i.e.*, ΔD and ΔV). We next study how to build HEV's such that eqid shipment is minimized.

Recall that HEV's and IDX's are used together to identify the equivalent classes of the input update (line 1 of both algorithms incVIns and incVDel in Fig. 4), whilst for each variable CFD $(X \rightarrow B, t_p[X])$, two IDX's must be built with the key eqid_X and $\mathsf{eqid}_{X\cup\{B\}}$ respectively for each input tuple , and HEV's are built to efficiently compute these keys for IDX's. As remarked earlier, how these HEV's are built decides how eqid's are shipped for generating the keys of IDX's. For multiple CFDs that may have common attributes, different orders on grouping attributes of HEV's may affect the number of eqid's shipped for an single update, as shown below.

Example 7: Consider a relation R_e with 11 attributes $A, B, \cdots, K$ that is vertically partitioned and distributed over 8 sites: $S_1(A)$, $S_2(B)$, $S_3(C)$, $S_4(D)$, $S_5(E, F)$, $S_6(G, H)$, $S_7(I)$, $S_8(J, K)$. Here $S_1(A)$ denotes that attribute A is at site S_1 (besides a key); similarly for the other attributes. A set Σ_e of CFDs is imposed on R_e, including $\varphi_1 : (ABC \rightarrow E)$, $\varphi_2 : (ACD \rightarrow F)$, $\varphi_3 : (AG \rightarrow H)$, and $\varphi_4 : (AIJ \rightarrow K)$.

Consider different HEV's for the CFDs in Fig. 6, in which a rectangle indicates a site, a circle an attribute, a triangle an HEV, an ellipse an IDX index, and a directed edge indicates an eqid shipment from one site to another. Note that one IDX is needed for each CFD. We omit those base HEV's that only used locally to simplify the figure.

(1) *No sharing between the* HEV*'s of different CFDs*. Figure 6(a) depicts a case when HEV's are independently built for the CFDs. These HEV's determine how eqid's are shipped when validating the CFDs. For example, when a tuple t is inserted into (or deleted from) R_e, to detect the violations of $\varphi_1 : (ABC \rightarrow E)$, we need to (a) identify the eqid of $t[A]$ from H_A at site S_1, which is shipped to S_2; (b) determine the eqid of $t[AB]$ from H_{AB} upon receiving the eqid of $t[A]$, which is in turn shipped to S_3; (c) detect the new violations (resp. removed violations) for inserting (resp. deleting) t by examining H_{ABC} and the IDX index *w.r.t.* φ_1 at site S_3. Two eqid's need to be shipped for φ_1. The process for the other CFDs is similar. In total, 9 eqid's (*i.e.*, the number of directed edges in Fig. 6(a)) need to be shipped to detect all violations of the CFDs in Σ_e. Note that when the eqid of $t[A]$ is shipped from S_1 to S_3, it is used by both H_{AC} (for φ_2) and H_{ABC} (for φ_1) at site S_3; hence this eqid is shipped only once.

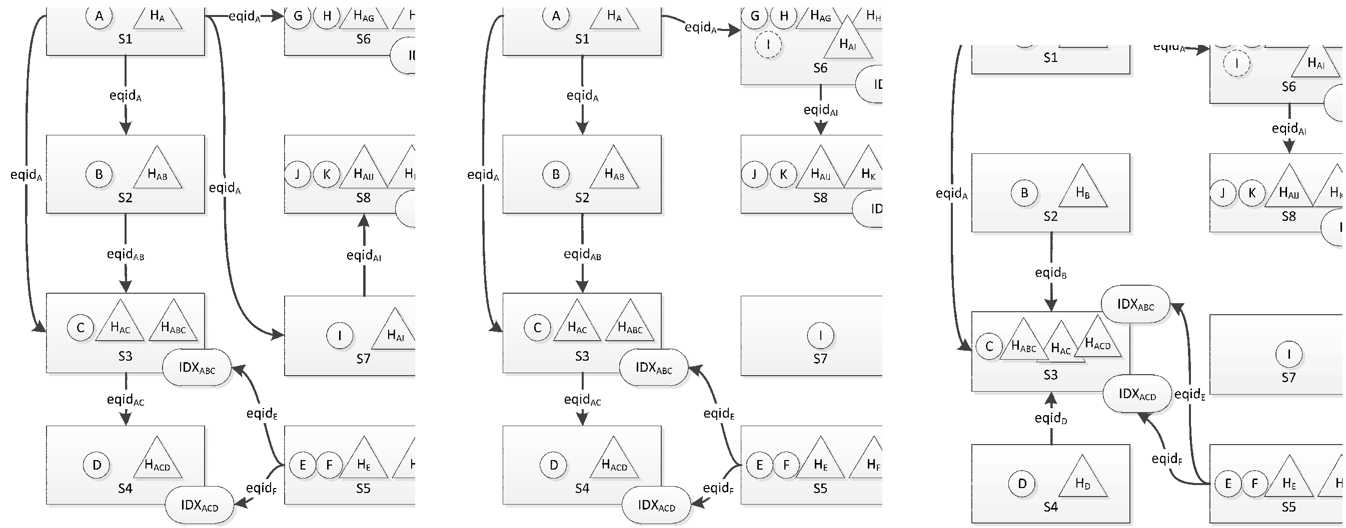
(2) *In the presence of replication*. Replication is common in distributed data management, to improve reliability and accessibility. Suppose that attribute I is replicated at site S_6 besides residing at S_7, as shown in Fig. 6(b). This allows us to choose either site S_6 or site S_7 where we build index H_{AI}, as opposed to Fig. 6(a) in which H_{AI} has to be built at S_7. Note that to detect the violations of $\varphi_3 : (AG \rightarrow H)$, the eqid for $t[A]$ needs to be shipped from S_1 to S_6 in both Fig. 6(a) and Fig. 6(b). If we build H_{AI} at S_6, we may send the eqid of $t[AI]$ from S_6 to S_8 (Fig. 6(b)), instead of from S_7 to S_8 (Fig. 6(a)) to validate $\varphi_4 = (AIJ \rightarrow K)$. This saves us one eqid shipment for $t[A]$ from S_1 to S_7 (Fig. 6(a)). In total, 8 eqid's need to be shipped in this case, instead of 9 in Fig. 6(a).

(3) *Sharing* HEV*'s among CFDs*. When I is replicated at site S_6, we can do better than Fig. 6(b), as depicted in Fig. 6(c). The key observation is that attributes AC are shared by CFDs φ_1 and φ_2. Hence, when a tuple t is inserted or deleted, we can compute the eqid of $t[AC]$ by shipping the eqid of $t[A]$ from S_1 to S_3. This allows us to compute the eqid's of $t[ABC]$ (with the eqid of $t[B]$ from S_2 to S_3) and $t[ACD]$ (with the eqid of $t[D]$ from S_4 to S_3) both at S_3 (Fig. 6(c)). In contrast, in the setting of Fig. 6(b) we have to compute eqid's by following the order of $t[A] \Rightarrow t[AB] \Rightarrow t[ABC]$ for φ_1 and $t[A] \Rightarrow t[AC] \Rightarrow t[ACD]$ for φ_2. In Fig. 6(c), only 7 eqid's need to be shipped as opposed to 8 eqid's in Fig. 6(b). □

Example 7 motivates us to find an optimal strategy for building HEV's, such that the keys of IDX's could be computed with minimum number of eqid shipments. It also suggests that we reduce eqid shipment by sharing HEV's among multiple CFDs as much as possible (*e.g.*, H_{AC} at S_3 for φ_1 and φ_2 in the case (3) above).

Below we first formalize this as an optimization problem, and show that it is NP-complete. We then provide an effective heuristic algorithm for building HEV's.

Optimization. A close look at the use of HEV in the detection algorithms and their complexity analysis (Section 4) reveals the following. To handle a unit update (insertion or deletion of a tuple t), the number of eqid's shipped is independent of (a) the values in database D and (b) the value of t. Indeed, eqid is shipped only when a non-base HEV needs eqid's generated from HEV's at



(a) No replication (b) With replication (c) Minimum eqid shipment

Fig. 6. Example of minimizing eqid shipment (base hash tables used only locally are omitted)

other sites, and hence, is decided by the dependencies between HEV's. Thus we can talk about eqid shipments for a unit update regardless of the values of D and t.

We show that the problem of building HEV's is already challenging for unit updates. Consider a schema R, a vertical partition scheme that partitions an instance D of R into $(D_1, \ldots, D_n)$ such that D_i resides at site S_i, and attributes of R may be replicated, *i.e.*, $(D_1, \ldots, D_n)$ may not be disjoint. Given a schema R, the partition and replication scheme for R, a set Σ of CFDs, and a positive number K, the *minimum* eqid *shipment problem* is to decide whether there exists a set $\mathcal{H}$ of HEV's such that for any instance D of R and any single update with tuple t, it needs no more than K eqid's shipped to find changes to $\mathsf{V}(\Sigma, D)$. Here for each $\varphi = (X \rightarrow B, t_p[X]) \in \Sigma$, $\mathcal{H}$ has to identify the keys eqid_X and $\mathsf{eqid}_{X\cup\{B\}}$ of two IDX's for φ, and it needs no more than K eqid shipments to find all such keys of IDX's for all CFDs in Σ.

Theorem 7: *The problem for minimum* eqid *shipment is NP-complete.* □

Proof. Upper bound. We show that the problem is in NP by giving an NP algorithm. It first guesses a set $\mathcal{H}$ of at most $\Sigma_{1\leq i\leq n}|R_i| + n*m$ hash tables with their locations, where $|R_i|$ is the number of attributes in partition D_i. Indeed, for each attribute in each D_i, one base hash table needs to be built (hence $\Sigma_{1\leq i\leq n}|R_i|$), and for each partition D_i and each CFD φ in Σ, we need at most 1 non-base hash table that contains all attributes of φ in D_i (hence $(m*n)$ non-base hash tables). After $\mathcal{H}$ is in place, we check (a) whether for any CFD $(X \rightarrow B, t_p[X]) \in \Sigma$, $\mathcal{H}$ can identify eqid_X and $\mathsf{eqid}_{X\cup\{B\}}$; and (b) whether we need no more than K eqid's shipped when validating all CFDs in Σ for a single update with tuple t. As remarked above, step (b) is independent of D and t. Steps (a) and (b) can be done by leveraging the dependencies between HEV's, in PTIME when the HEV's and their locations are given. If the number of eqid shipments is no more than K via $\mathcal{H}$, then $\mathcal{H}$ provides the indices we need. Otherwise we guess another $\mathcal{H}$ and repeat the process. This algorithm is in NP, and hence so is the problem.

Lower bound. We next show that problem is NP-hard by reduction from the minimum set cover problem (MSC; see the proof of Theorem 3 for the statement of MSC).

Given an instance $(X, \mathcal{C}, K)$ of MSC, we construct (R, Σ, K) such that the minimum eqid shipment problem for (R, Σ, K) has a solution iff the MSC problem has a solution. Assume *w.l.o.g.* that $X = \{x_j \mid j \in [1, m]\}$, $\mathcal{C} = \{C_i \mid i \in [1, n]\}$, each C_i has three elements of X, and that $X = \bigcup_{i\in[1,n]} C_i$ (*i.e.*, there exists a cover for X).

(a) We define a schema $R = (\mathsf{id}, Y, Z, X_1, X_2, \ldots X_m)$, a partition and replication scheme that vertically partition any instance D of R into $n+1$ fragments $U, D_1, D_2, \ldots, D_n$, with schemas $R_U = (\mathsf{id}, Y)$ for U and $R_i = (\mathsf{id}, Z, X_{a_1}, X_{a_2}, X_{a_3})$ for D_i. Here x_{a_1}, x_{a_2} and x_{a_3} are elements in $C_i \in \mathcal{C}$. Intuitively, each D_i encodes a set C_i. and attributes may be duplicated in different sites.

(b) The set Σ consists of m FDs: $X_1Y \rightarrow Z$, $X_2Y \rightarrow Z$, $\ldots$, and $X_mY \rightarrow Z$. Intuitively, each $X_iY \rightarrow Z$ encodes the element x_i in X. Thus the set Σ encodes the set X.

We show that (R, Σ, K) is a reduction from MSC. First, assume that the MSC instance has a cover C' of size no larger than K. We define a set $\mathcal{H}$ as follows.

(a) On each site S_i, where $C_i = \{x_{a_1}, x_{a_2}, x_{a_3}\} \in C'$, $\mathcal{H}$ has the following HEV's: (i) $(h_{i0} : Z \rightarrow \mathsf{eqid}_Z)$; (ii) $(h_{i1} : X_{a_1} \rightarrow \mathsf{eqid}_{X_{a_1}})$, $(h_{i2} : X_{a_2} \rightarrow \mathsf{eqid}_{X_{a_2}})$, and $(h_{i3} : X_{a_1} \rightarrow \mathsf{eqid}_{X_{a_3}})$; (iii) $(h'_{i1} : \mathsf{eqid}_{X_{a_1}}, \mathsf{eqid}_Y \rightarrow \mathsf{eqid}_{X_{a_1}Y})$, $(h'_{i2} : \mathsf{eqid}_{X_{a_2}}, \mathsf{eqid}_Y \rightarrow \mathsf{eqid}_{X_{a_2}Y})$, and $(h'_{i3} : \mathsf{eqid}_{X_{a_3}}, \mathsf{eqid}_Y \rightarrow \mathsf{eqid}_{X_{a_3}Y})$; (iv) $(h''_{i1} : \mathsf{eqid}_{X_{a_1}}, \mathsf{eqid}_Y, \mathsf{eqid}_Z \rightarrow \mathsf{eqid}_{X_{a_1}YZ})$, $(h''_{i2} : \mathsf{eqid}_{X_{a_2}}, \mathsf{eqid}_Y, \mathsf{eqid}_Z \rightarrow \mathsf{eqid}_{X_{a_2}YZ})$, and $(h''_{i3} : \mathsf{eqid}_{X_{a_3}}, \mathsf{eqid}_Y, \mathsf{eqid}_Z \rightarrow \mathsf{eqid}_{X_{a_3}YZ})$.

(b) On the site S_U, $\mathcal{H}$ includes $(h_U : Y \rightarrow \mathsf{eqid}_Y)$.

Intuitively, to check a unit update t posed on any instance D of R, it suffices to ship the eqid_Y for t generated by (b) from S_U to S_i for each $C_i \in C'$. In total $|C'|$ eqid's are shipped (see the algorithms in Section 4). Indeed, since C' is a cover for X and Σ encodes X, one can verify the following: HEV's in (a)(iii) (resp. (a)(iv)) generate all eqid_{X_iY} (resp. eqid_{X_iYZ}) for each FD $(X_iY \rightarrow Z) \in \Sigma$, and all eqid's required for (a)(iii) and (a)(iv) are provided by eqid shipments of (c) for tuple t. Hence $\mathcal{H}$ suffices to generate all the eqid's needed by Σ. Since $|C'| \leq K$, the number of eqid shipments via $\mathcal{H}$ is at most K.

Conversely, assume that there exists a set $\mathcal{H}$ of hash tables such that for any FD $(X_iY \rightarrow Z) \in \Sigma$, $\mathcal{H}$ can find eqid_X and $\mathsf{eqid}_{X\cup\{B\}}$, and moreover, for any D and unit update with a tuple t, the number of eqid's shipped for computing eqid's of all CFDs in Σ is at most K. Consider the following cases. (a) If $K \geq n$, the set $\mathcal{C}$ is a cover and $|\mathcal{C}| = n \leq K$. (b) If $K < n$, let C' consist of those C_i's such that eqid's are shipped between U and D_i ($i \in [1, n]$) of $\mathcal{H}$ when handling the update. One can verify that $|C'| \leq K$ and C' is a cover for X, since otherwise, there must exist an uncovered element x_j in X such that eqid_{X_jY} for t could not be generated and checked. □

Due to the intractability, any efficient algorithm to find an optimal plan to build HEV's is necessarily heuristic.

A heuristic algorithm. We next provide an efficient heuristic algorithm for building HEV's. The idea behind the algorithm is to start with HEV's with the keys for IDX's. That is, for a CFD $\varphi = (X_\varphi \rightarrow Y_\varphi,\ t_{p_\varphi})$, we first build an HEV for X_φ, which is necessary for detecting violations of φ. We then build HEV's for certain subsets of X_φ, by selecting those subsets that contain as many attributes shared by multiple CFDs as possible. We also include base HEV's that contain attributes that only reside at one site, *e.g.,* H_A at site S_1 in Fig. 6(a), since H_{AB} at S_2 requires H_A at S_1 and local attribute B at S_2 as input, while H_{AB} at site S_2 in Fig. 6(a)) is not. Finally, we remove redundant HEV's while ensuring that all violations can still be detected. It follows a greedy approach that determines the key (set of eqid's) of each HEV and retains the HEV's with the minimum eqid shipment among the solutions explored. It terminates when no more HEV can be removed.

The algorithm, referred to as optVer, is shown in Fig. 7. It takes as input a database D that is vertically partitioned into D_i (for $i \in [1, n]$) and allows a predefined replication scheme, a set Σ of CFDs, and a parameter k for balancing the effectiveness and efficiency. It builds a set $\mathcal{H}$ of HEV's for Σ. The algorithm works as follows.

(1) [Initialization.] It builds a set $\mathcal{H}$ of HEV's such that for each $\varphi \in \Sigma$, there is an HEV with key X_φ (lines 1-4).

(2) [Expansion.] It then expands $\mathcal{H}$. For each CFD φ, we add up to $|\Sigma|+|X_\varphi|$ HEV's, by including the HEV's whose keys contain as many attributes shared by multiple CFDs as possible (lines 5-6). For each attribute of each CFD in Σ, we also build a base HEV (line 7), such that all existing HEV's can take their outputs and compute eqid's.

(3) [Location.] We assign a site to each HEV h in $\mathcal{H}$ (line 8).

Algorithm optVer
Input: D in n vertical partitions, a set Σ of CFDs, a parameter k
Output: a set $min_{\mathcal{H}}$ of HEV's.
1. $\mathcal{H} := \emptyset$;
2. **for each** $\varphi \in \Sigma$ **do** /* $\varphi : (X_\varphi \rightarrow Y_\varphi, t_{p_\varphi})$ */
3. $\quad \mathcal{H} := \mathcal{H} \cup \{$an HEV for $X_\varphi\}$;
4. $\mathcal{H}_{\mathsf{IDX}} := \mathcal{H}$; /* *HEV's that are necessary for IDX's* */
5. **for each** $\varphi \in \Sigma$ **and** $\phi \in \Sigma \setminus \{\varphi\}$ **do** $\mathcal{H} := \mathcal{H} \cup \{$an HEV for $X_\varphi \cap X_\phi\}$;
6. **for each** $\varphi \in \Sigma$ **do** add up to $|X_\varphi|$ HEV's having shared attributes;
7. Expand $\mathcal{H}$ with necessary base HEV's;
8. **for each** $h \in \mathcal{H}$ **do** h.location := findLoc(h);
   /* *min and $min_{\mathcal{H}}$ keep the best solution so far; $\mathcal{H}.N_{\mathsf{eqid}}()$ returns #-eqid shipments for $\mathcal{H}$; Q is the queue for* BFS */
9. $min := \mathcal{H}.N_{\mathsf{eqid}}()$; $\quad min_{\mathcal{H}} := \mathcal{H}$; $\quad Q := \{\mathcal{H}\}$;
10. **while** ($Q \neq \emptyset$) **do**
11. $\quad Q' := \emptyset$;
12. $\quad$ **while** ($\mathcal{H}$ = Q.pop()) **do**
13. $\quad\quad$ **if** $min > \mathcal{H}.N_{\mathsf{eqid}}()$ **then** $min := \mathcal{H}.N_{\mathsf{eqid}}()$; $\quad min_{\mathcal{H}} := \mathcal{H}$;
14. $\quad\quad$ **for each** $h \in \mathcal{H}$ **do**
15. $\quad\quad\quad$ **if** all HEV's in $\mathcal{H}_{\mathsf{IDX}}$ are computable by $(\mathcal{H} \setminus \{h\})$ **then**
16. $\quad\quad\quad\quad Q'$.push($\mathcal{H} \setminus \{h\}$);
17. $\quad$ Keep up to k distinct $\mathcal{H}$'s with smallest $\mathcal{H}'.N_{\mathsf{eqid}}()$ in Q';
18. $\quad Q := Q'$;
19. **return** $min_{\mathcal{H}}$;

Fig. 7. Heuristic algorithm for minimizing eqid shipment

The site is determined by findLoc, such that (a) the local attributes at the site cover as many attributes of h as possible, and (b) as many other HEV's reside at the site as possible. This takes into account of the replication.

(4) [Finalization.] We follow a greedy approach to searching an optimal solution by removing HEV's from $\mathcal{H}$ (lines 9-18). After steps (2)–(4), some tables in $\mathcal{H}$ may be redundant, *i.e.,* unnecessary for computing those tables needed by IDX's ($\mathcal{H}_{\mathsf{IDX}}$). We iteratively remove HEV's from $\mathcal{H}$ until removing any more table will make some HEV in $\mathcal{H}_{\mathsf{IDX}}$ no longer computable (lines 10-18). In the process we record the best solution so far in $min_{\mathcal{H}}$ (line 13). More specifically, we conduct search in the BFS fashion: each state is a set of HEV's, Q keeps all open states, and the algorithm only includes the top k solutions (measured by the number of eqid shipped) in Q in each iteration (line 17), where k is a user defined threshold to balance the effectiveness and efficiency.

The function $\mathcal{H}.N_{\mathsf{eqid}}()$ computes the number of eqid shipments for a given set $\mathcal{H}$ of HEV's. It also determines the order and structure of each HEV h as follows: at each stage, it selects an HEV h' from $\mathcal{H}$ whose key attributes contain the largest number of uncovered attributes in h. The eqid computed from h' is to be shipped to h.

Example 8: Consider the data partition of Fig. 6(c) described in Example 7, where I is replicated at S_6. Taking these as input, optVer builds HEV's as follows.

(1) [Initialization.] It first builds 4 HEV's H_{ABC}, H_{ACD}, H_{AG} and H_{AIJ}, for CFDs φ_1, φ_2, φ_3, and φ_4, respectively.

(2) [Expansion.] It adds the following tables:

(a) H_A, since A is shared by all CFDs, and H_{AC}, as attributes AC are shared by φ_1 and φ_2;

(b) H_{AI} and H_{AJ}, in which keys are subsets of X_{φ_4}, and both contain attribute A; and

(c) base HEV for the CFDs in Σ_e: H_B, …, H_J, H_K.

(3) [Location.] It assigns a site for each HEV to reside at: H_{ABC}, H_{ACD} at S_3, H_{AG} at S_6, and H_{AIJ} at S_8; each base HEV is located at the site where its attribute is located (*e.g.*, H_A at S_1 and H_B at S_2).

(4) [Finalization.] Assume that $k = 5$, it removes redundant H_{AJ}. The solution of Fig. 6(c) is then found, with 7 eqid's shipped in total. □

Complexity. The algorithm is in $O(k|\Sigma|^4 + n|\Sigma|)$ time. Indeed, it takes $O(k|\Sigma|^4)$ time for the iterations (lines 9–18) and $O(n|\Sigma|)$ time for site assignments (line 8). More specifically, the outer **while** iteration is bounded by the number of HEV's in $\mathcal{H}$ (*i.e.*, $O(|\Sigma|^2)$), the inner **while** iterates at most k times for each outer **while** iteration, the inner **for** loop runs at most $|\Sigma|^2$ times, and $N_{\mathsf{eqid}}()$ inside the **for** loop could be computed in $O(1)$ time using proper dynamic programming techniques. For other steps, it is in $O(|\Sigma|)$ time for lines 1-4, $O(|\Sigma|^2)$ time for line 5, and in $O(|\Sigma|^2)$ time for lines 6-7. Note that the number of rules $|\Sigma|$ is usually small in practice, and the algorithm only needs to be run once for given database D, replication scheme, and CFDs Σ instead of each time calling optVer at each update.

6 Algorithms for Horizontal Partitions

When it comes to horizontal partitions, there also exist incremental detection algorithms that are *optimal*.

Proposition 8: *There exists an algorithm that incrementally detects CFD violations in horizontal partitions with communication and computational costs in* $O(|\Delta D| + |\Delta \mathsf{V}|)$. □

Taken together, Propositions 6 and 8 verify Theorem 5.

Along the same lines as its vertical counterpart, we first identify when data shipment can be avoided. We then give an optimal algorithm for horizontal partitions.

Consider a database $D = (D_1, \ldots, D_n)$ that is horizontally partitioned, where D_i resides at site S_i for $i \in [1, n]$.

Local checking. For horizontal partitions, CFDs that can be validated locally include the following.

(1) Constant CFDs. Such a CFD can be violated by a *single* tuple, and does not incur global violations. Hence no data shipment is needed for validating constant CFDs.

(2) Variable CFDs. Notably, a horizontal fragment D_i is defined as $\sigma_{F_i}(D)$ (Section 2). We use X_{F_i} to denote all attributes in F_i. To validate a variable CFD $\phi = (X \rightarrow B, t_p)$, one does not have to ship data to or from S_i when

- (a) $X_{F_i} \subseteq X$; indeed, for any tuple $t \in D_i$ and $t' \notin D_i$, (t, t') do not violate ϕ since $t[X_{F_i}] \neq t'[X_{F_i}]$; or
- (b) $F_i \wedge F_\phi$ evaluates to false [10], where F_ϕ is a conjunction of atoms $A =$ 'a' imposed by t_p, for $A \in X$. Indeed, no tuples in D_i could possibly match $t_p[X]$.

Algorithms. We first consider a single CFD and a single update. We then extend the algorithm to multiple CFDs and batch updates. At each site, we also maintain the indices (*only* for local tuples) for equivalence classes and set() similar to the ones introduced in Section 4.

Single update for one CFD. Given a CFD $\phi = (X \rightarrow B, t_p)$ and a tuple t to be inserted into (resp. deleted from) D_i, the algorithm is to identify the changes $\Delta\mathsf{V}^+(\phi, D)$ (resp. $\Delta\mathsf{V}^-(\phi, D)$) to $\mathsf{V}(\varphi, D)$, outlined below.

Insertions. The algorithm handles insertions as follows.

(1) Site S_i checks local violations. It deals with two cases:

(a) There exist no local violations, *i.e.*, there is no $t' \in D_i$ such that $(t, t') \not\models \phi$. Then there are again two cases:

- (i) when $[t]_{X\cup\{B\}} \neq \emptyset$: $\Delta\mathsf{V}_i^+ = \{t\}$ if $|\mathsf{set}(t[X])| > 1$, and $\Delta\mathsf{V}_i^+ = \emptyset$ otherwise; indeed, if $t' \in [t]_{X\cup\{B\}}$ is a known violation, so is t; or neither is a violation; and
- (ii) when $[t]_{X\cup\{B\}} = \emptyset$: we need to send t to other sites to check global violations, *i.e.*, to find out whether there exists a tuple $t' \notin D_i$ such that $(t, t') \not\models \phi$. We set $\Delta\mathsf{V}_i^+ = \{t\}$ if such t' exists, and $\Delta\mathsf{V}_i^+ = \emptyset$ otherwise.

(b) Local violations exist, *i.e.*, there exists $t' \in D_i$ such that $(t, t') \not\models \phi$. We consider the following two cases:

- (i) when $[t]_{X\cup\{B\}} \neq \emptyset$: then $\Delta\mathsf{V}_i^+ = \{t\}$, since any tuple that violates ϕ with t is a known violation; and
- (ii) when $[t]_{X\cup\{B\}} = \emptyset$: then there must exist a tuple $t' \in D_i$ such that $(t, t') \not\models \phi$. If $t' \in \mathsf{V}_i$, we have $\Delta\mathsf{V}_i^+ = \{t\}$; otherwise $\Delta\mathsf{V}_i^+ = \{t\} \cup [t']_{X\cup\{B\}}$ since each tuple in $[t']_{X\cup\{B\}}$ violates ϕ with t. In both cases, we need to check global violations by sending t to all the other sites, which check violations incurred by inserting tuple t.

(2) Upon receiving t from S_i, each site S_j ($j \neq i$) checks its local violations *in parallel*, as described in step 1(a).

The global changes $\Delta\mathsf{V}^+$ is the union of changed violations from all the sites, *i.e.*, $\Delta\mathsf{V}^+ = \bigcup_{k\in[1,n]} \Delta\mathsf{V}_k^+$.

Deletions. When a tuple t is deleted from D_i at Site S_i, the algorithm does the following at S_i and other sites.

(1) *At site* S_i. It first identifies $[t]_{X\cup\{B\}}$ and $\mathsf{set}(t[X])$ at S_i for CFD φ. If t does not violate ϕ, then t is simply deleted from D_i, since deletions do not introduce new violations. When t violates ϕ, there are two cases to consider.

(a) If after t is deleted, tuples that agree with t on both X and B remain, then all violations except t remain.

(b) Otherwise, the entire entry for t will be removed. There are again two cases to consider:

- (i) There are two items in $\mathsf{set}(t[X])$, t and t'. It broadcasts t' to the sites that have violations with t or t'. We record the sites that still have violations. It removes all violations *w.r.t.* t and t' if no sites have tuples that violate t', and otherwise only t is removed from violations.
- (ii) Tuple t is the only entry at site S_i. It removes t as a violation, and broadcasts t to the other sites that previously have violations with t.

The local index is maintained and $\Delta\mathsf{V}_i^-$ is then returned.

(2) At site S_j. Upon receiving t from S_i, each site S_j ($j \neq i$) checks whether previous violations maintained

Algorithm incHor
Input: ΔD, D in n horizontal partitions, Σ, and $\mathsf{V}(\Sigma, D)$.
Output: $\Delta\mathsf{V}$.
1. merge local updates in ΔD_i having the same tuple ids;
2. $\Delta\mathsf{V}^- := \emptyset$; $\Delta\mathsf{V}^+ := \emptyset$;
3. **for each** $\phi \in \Sigma$ **do**
4. **if** ϕ is a constant CFD **then** /* $\phi = (X \rightarrow B, t_p)$ */
5. **for each** $t \in \Delta D_i$ $(i \in [1, n])$ **and** t violates ϕ **do**
6. **if** $t \in \Delta D_i^-$ **then** $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \{t\}$;
7. **elseif** $t \in \Delta D_i^+$ **then** $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \{t\}$;
8. **elseif** ϕ can be locally checked at S_i **then**
9. derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ at S_i with indices (Section 6);
10. $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$; $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$
11. **else** /* a variable CFD that cannot be *locally checked* */
12. derive $\Delta\mathsf{V}_i^+$ and $\Delta\mathsf{V}_i^-$ $(i \in [1, n])$;
13. $\Delta\mathsf{V}^- := \Delta\mathsf{V}^- \cup \Delta\mathsf{V}_i^-$ **and** $\Delta\mathsf{V}^+ := \Delta\mathsf{V}^+ \cup \Delta\mathsf{V}_i^+$ $(i \in [1, n])$;
14. **return** $\Delta\mathsf{V} = \Delta\mathsf{V}^- \cup \Delta\mathsf{V}^+$;

Fig. 8. Batch updates for horizontal partitions

at S_j could be removed. Note that S_j will send two different messages: either (a) t' from S_i ((1)(b)(i) above): this means that t' remains at S_i; or (b) t from S_i ((1)(b)(ii) above): this means that t is removed from S_i.

The global changes $\Delta\mathsf{V}^-$ is the union of $\Delta\mathsf{V}_k^-$ $(k = [1, n])$, from all individual sites.

Example 9: Consider D_0 (without t_6) given in Fig. 2 and ϕ_1 of Fig. 1. When tuple t_6 is inserted, the algorithm finds that $(t_6, t_5) \not\models \phi_1$ at site S_3 (step (1)(a)), *i.e.*, no local violations. However, since t_5 is a known violation (Fig. 1), so is t_6 (step (1)(a)(i)). Hence, $\Delta\mathsf{V}^+ = \{t_6\}$. □

Batch updates and multiple CFDs. We now present an algorithm for batch updates and multiple CFDs on horizontal partitions, denoted as incHor and shown in Fig. 8. Given batch updates ΔD, a horizontal partition $(D_1, \cdots, D_n)$ of a database D, a set Σ of CFDs, and (old) violations $\mathsf{V}(\Sigma, D)$ of Σ in D, the algorithm finds and outputs the changes $\Delta\mathsf{V}$ to violations $\mathsf{V}(\Sigma, D)$.

The algorithm first removes the local updates that cancel each other (line 1), and initializes the changes (line 2). It then detects the changes to violations for multiple CFDs in parallel (lines 3-13). It deals with three cases as follows. (1) *Constant* CFDs (lines 4-7). It checks at each site that whether a deletion removes a violation (line 6) or an insertion adds a violation (line 7). (2) *Locally checked variable* CFDs (lines 8-10). The changes to violations can be detected using the same indices as used in Section 4, in constant time (lines 9-10). (3) *General variable* CFDs (lines 11-13). The changes to violations are identified (lines 12-13), and then returned (line 14).

Complexity. For communication cost, one can see that each tuple in ΔD is sent to other sites at most once. Hence at most $O(|\Delta D|\ n)$ messages are sent, where n is the number of fragments and is fixed, as remarked earlier. Thus the cost is in $O(|\Delta D|)$. The computation cost is in $O(|\Sigma|(|\Delta D| + |\Delta\mathsf{V}|))$ time, where $|\Sigma|$ is a fixed parameter. That is, it is in $O(|\Delta D| + |\Delta\mathsf{V}|)$. Indeed, by leveraging hash tables, the process at each site takes constant time, and the hash tables can be maintained incrementally in the same process, also in constant time.

Optimization using MD5. A tuple may be large. To reduce its shipping cost, a natural idea is to encode the whole tuple, and then send the coding of the tuple instead of the tuple. MD5 (Message-Digest algorithm 5 [1]) is a widely used cryptographic hash function with a 128-bit hash value. We use MD5 in our implementation to further reduce the communication cost, by sending a 128-bit MD5 code instead of an entire tuple.

7 Experimental Study

We present an experimental study of our incremental algorithms for vertical and horizontal partitions, evaluating elapsed time and data shipment. We focus on their scalability by varying four parameters: (1) $|D|$: the size of the base relation; (2) $|\Delta D|$: the size of updates; (3) $|\Sigma|$: the number of CFDs; and (4) n: the number of partitions. We also evaluated the effectiveness of our optimization techniques for building indices in vertical partitions.

Experimental setting. We used the following datasets.

(1) Datasets. (a) TPCH: we joined all tables to build one table. The data ranges from 2 million tuples (*i.e.*, 2M) to 10 million tuples (*i.e.*, 10M). Notably, the size of 10M tuples is **10GB**. (b) DBLP: we extracted a 320MB relation from its XML data. It scales from 100K to 500K tuples.

(2) CFDs were designed manually. We first designed functional dependencies (FDs), and then produced CFDs by adding patterns (*i.e.*, conditions) to the FDs. For TPCH: the number $|\Sigma|$ of CFDs ranges from 25 to 125, with increment of 25 by default. For DBLP: $|\Sigma|$ scales from 8 to 40, with increment of 8 by default.

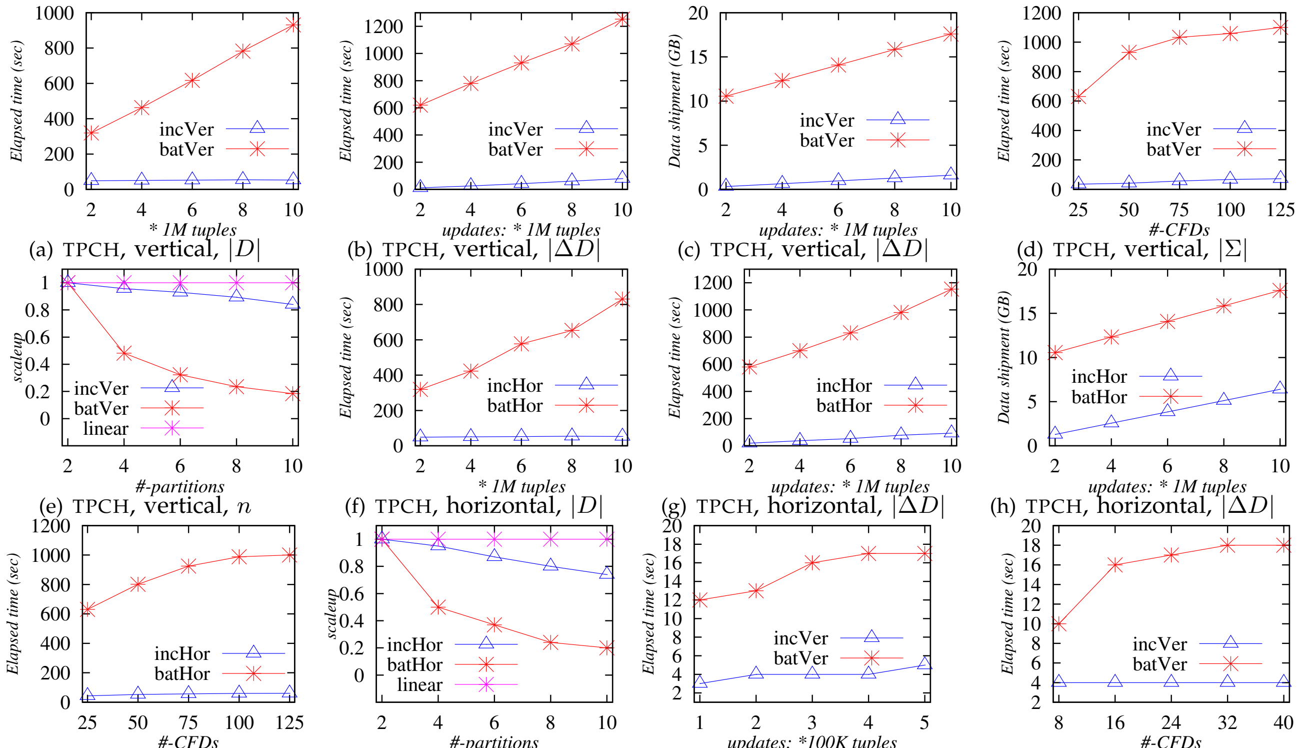
(3) Updates. Batch updates contain 80% insertions and 20% deletions, since insertions happen more often than deletions in practice. The size of updates is up to 10M tuples (about **10GB**) for TPCH and up to 320MB for DBLP.

(4) Partitions. Its fragment number is 10 by default.

Implementation. We denote by incVer (resp. incHor) our incremental algorithms for batch updates and multiple CFDs in vertical (resp. horizontal) partitions. We also designed batch algorithms for detecting errors in vertical (resp. horizontal) partitions, denoted by batVer (resp. batHor), following [10]. The batch algorithms work in three steps: (1) for each CFD it copies to a coordinator site a small number of relevant attributes (resp. tuples) for vertical (resp. horizontal) partitions; (2) the violations of each CFD ϕ are checked locally at the coordinator site for ϕ; and (3) the violations of all CFDs are checked in parallel. All algorithms were written in Python. We ran our experiments on Amazon EC2 High-Memory Extra Large instances (zone: us-east-1c).

In the following, we shall pay more attention to TPCH, more interesting for its larger size than DBLP.

Experimental results for vertical partitions. We first present our experimental results of detecting violations in data that is vertically partitioned and distributed.



(a) TPCH, vertical, $|D|$ (b) TPCH, vertical, $|\Delta D|$ (c) TPCH, vertical, $|\Delta D|$ (d) TPCH, vertical, $|\Sigma|$
(e) TPCH, vertical, n (f) TPCH, horizontal, $|D|$ (g) TPCH, horizontal, $|\Delta D|$ (h) TPCH, horizontal, $|\Delta D|$
(i) TPCH, horizontal, $|\Sigma|$ (j) TPCH, horizontal, n (k) DBLP, vertical, $|\Delta D|$ (l) DBLP, vertical, $|\Sigma|$

Fig. 9. Experimental results for TPCH and DBLP data

Exp-1: Impact of $|D|$. Fixing $|\Delta D| = 6$M, $|\Sigma| = 50$ and $n = 10$, we varied the size of D (*i.e.*, $|D|$) from 2M to 10M tuples (10GB) for TPCH. Figure 9(a) shows the elapsed time in seconds when varying $|D|$. The result tells us that incVer outperforms batVer by two orders of magnitude. It also shows that the elapsed time of incVer is insensitive to $|D|$. In contrast, the elapsed time of batVer increases much faster when $|D|$ is increased. This result further verifies Proposition 6: the incremental algorithm is bounded by the size of the changes in the input and output, and it is independent of D.

Exp-2: Impact of $|\Delta D|$. Fixing $|\Sigma| = 50$, $n = 10$ and $|D| = 10$M, we varied the size of ΔD from 2M to 10M tuples for TPCH. We also varied $|\Delta D|$ from 100K to 500K tuples for DBLP while fixing $|D| = 500$K, $|\Sigma| = 16$ and $n = 10$.

Figure 9(b) (resp. Figure 9(k)) shows the elapsed time in seconds when varying $|\Delta D|$ for TPCH (resp. DBLP). Both figures show that the elapsed time of incVer increases almost linearly with $|\Delta D|$, *e.g.*, 11 seconds when $|\Delta D| = 2$M and 79 seconds when $|\Delta D| = 10$M as shown in Fig. 9(b). In addition, batVer is slower than incVer by two orders of magnitude, consistent with Fig. 9(a).

In addition, Figure 9(c) shows the size of data shipped (in GB) when varying $|\Delta D|$ for TPCH. Note that incVer only sends 320MB when $|\Delta D| = 2$M (*i.e.*, 2GB) and 1.6GB when $|\Delta D| = 10$M (*i.e.*, 10GB). This is because with HEVs, we only ship eqid's instead of the entire tuples. In contrast, the size of data shipped for batVer is up to 17.6GB when $|\Delta D| = 10$M. This further verifies our observation from Figure 9(b).

These experimental results tell us that our incremental methods are bounded by $|\Delta D| + |\Delta V|$, independent of the size of D, in contrast to batch algorithms that detect violations starting from scratch, which depends on $|D|$.

Exp-3: Impact of $|\Sigma|$. Fixing n=10, $|D|$=10M and $|\Delta D|$= 6M for TPCH, we varied $|\Sigma|$ from 25 to 125. Fixing n=10, $|D|$=500K and $|\Delta D|$=300K for DBLP, we varied $|\Sigma|$ from 8 to 40. Figure 9(d) (resp. Figure 9(l)) shows the elapsed time when varying $|\Sigma|$ from 25 to 125 for TPCH (resp. from 8 to 40 for DBLP). Both figures show that incVer achieves almost linear scalability when varying $|\Sigma|$, *e.g.*, 35 seconds when $|\Sigma|$=25 and 72 seconds when $|\Sigma|$=125 in Fig. 9(d). When multiple CFDs are detected, multiple sites work in parallel to improve the efficiency. Moreover, batVer runs far slower than incVer, as expected.

The results demonstrate that incVer scale well with $|\Sigma|$, and it can handle a large number of CFDs. We remark that in practice, Σ is typically predefined and fixed.

Exp-4: Impact of n. In this set of experiments, we varied the number of partitions from 2 to 10, and varied $|D|$ and $|\Delta D|$ in the same scale correspondingly. That is, we varied both $|D|$ and $|\Delta D|$ from 2M to 10M for TPCH. We study the *scaleup* performance defined as follows:

$$\text{scaleup} = \frac{\text{small system elapsed time on small problem}}{\text{large system elapsed time on large problem}}$$

Scaleup is said to be *linear* if it is 1, the ideal case.

Figure 9(e) shows the scaleup performance when varying n, $|D|$ and $|\Delta D|$ at the same time, where x-axis represents n and y-axis the scaleup value. The line for *linear* is the ideal case. For example, we computed the scaleup when $n = 4$ as follows: using the elapsed time when $n = 2$ and $|D| = |\Delta D| = 2$M to divide the elapsed time when $n = 4$ and $|D| = |\Delta D| = 4$M tuples (*i.e.*, 4GB in size), which is 0.96; similarly for all the other

Dataset	without optimization #-eqid shipments	with optimization #-eqid shipments
TPCH	122	55
DBLP	61	17

Fig. 10. Number of eqid's shipped for vertical partitions

points. This figure shows that incVer achieves nearly *linear* scaleup, which clearly outperforms batVer that shows bad scaleup performance.

These results indicate that incVer scales well with partitions, when base data and updates are large.

Optimization for vertical partitions. We next evaluate the effectiveness of our optimization strategy (Section 5).

Exp-5. Figure 10 shows the number of eqid's shipped for vertically partitioned TPCH (D = 10M, $|\Sigma| = 50$, and $n = 10$) and DBLP (D = 500K, $|\Sigma| = 16$, and $n = 10$), with or without using the optimization methods presented in Section 5. As remarked earlier, for each tuple insertion or deletion, the amount of eqid's shipped is independent of $|D|$. The table tells us that for both datasets, the optimization technique significantly reduces the number of eqid's to be shipped: it saves 67 eqid's (55.5%) for TPCH and 44 eqid's (72.1%) for DBLP per update.

Experimental results for horizontal partitions for TPCH. We next present results on horizontally partitioned data.

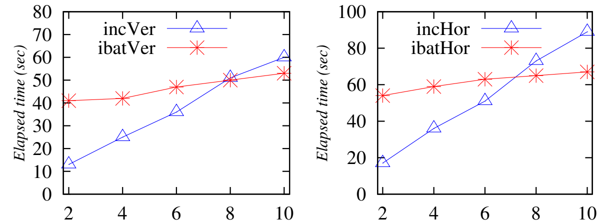
Exp-6: Impact of $|D|$. We adopted the same setting as Exp-1. Figure 9(f) shows the elapsed time when varying $|D|$. Besides telling us that incHor outperforms batHor, the results also show that incHor is independent of D: when varying $|D|$ from $2M$ to $10M$ tuples, the time only changes slightly. This verifies Proposition 8: incremental violation detection in horizontal partitions depends only on $|\Delta D|$ and $|\Delta \mathrm{V}|$, and is independent of D.

Exp-7: Impact of $|\Delta D|$. We used the same setting as Exp-2. Figure 9(g) shows the elapsed time when varying $|\Delta D|$ for TPCH. The results show that incHor increases almost linearly with the size of ΔD, *e.g.*, 19 seconds when $|\Delta D| = 2M$ and 93 seconds when $|\Delta D| = 10M$. Figure 9(h) shows the size of data shipment for both methods. The results verify that our incremental detection algorithm for horizontal partitions is bounded by $|\Delta D|$, similar to its vertical counterpart (see Exp-2).

Exp-8: Impact of $|\Sigma|$. We adopted the same setting as Exp-3. Figure 9(i) shows the elapsed time when varying $|\Sigma|$ from 25 to 125. It tells us that incHor is almost linear in $|\Sigma|$, *e.g.*, 43 seconds when $|\Sigma| = 25$ and 61 seconds when $|\Sigma| = 125$. The results verify that incHor scales well with $|\Sigma|$, as its vertical counterpart (see Exp-3).

Exp-9: Impact of n. Figure 9(j) shows the scaleup performance of incHor when varying n, $|D|$ and $|\Delta D|$ in the same scale, where x-axis represents the number n of fragments and y-axis the scaleup values. From the results we can see that incHor has nearly ideal scaleup, as its vertical counterpart. This verifies that our algorithms can work well on massive data, updates, and partitions.

Exp-10. Algorithms incVer and incHor substantially outperform existing batch algorithms. To favor the batch approach, we improved the batch algorithms, denoted by ibatVer and ibatHor for vertical and horizontal partitions, respectively, by using our incremental insertion algorithms and indices. We evaluated the performance of incVer and incHor vs. ibatVer and ibatHor starting with $\emptyset$, and inserting and deleting tuples until it reaches D.



(a) **Ver**, $|\Delta D|$: $*$1M tuples (b) **Hor**, $|\Delta D|$: $*$1M tuples

Fig. 11. Experimental results for refined batch algorithms

Figure 11(a) (resp. Figure 11(b)) shows the result for vertical (resp. horizontal) partition when $|D|$ = 6M, $|\Sigma| = 50$ and $n = 10$, while varying $|\Delta D|$ from 2M to 10M with 40% deletions and 60% insertions. The performance of batVer and batHor is not shown, since they are two orders of magnitude slower. The results tell us that in both vertical and horizontal partitions, the incremental algorithms do better than the revised batch algorithms until updates ΔD get rather large, *e.g.*, $|\Delta D|$ = 8M for vertical partitions and 7.6M for horizontal partitions.

Summary. From the experimental results we find the following. (1) Our incremental algorithms scale well with $|D|$, $|\Delta D|$ and $|\Sigma|$ for both vertical partitions (Exp-1 to Exp-4) and horizontal partitions (Exp-6 to Exp-9). (2) The incremental algorithms outperform their batch counterparts by two orders of magnitude, for reasonably large updates. But when updates are very large, batch algorithms do better, as expected (Exp-10). (3) The optimization techniques of Section 5 substantially reduce data shipment for vertical partitions (Exp-5). We contend that these incremental methods are promising in detecting inconsistencies in large-scale distributed data, for both vertically and horizontally partitioned data.

8 Conclusion

We have studied incremental CFD violation detection for distributed data, from complexity to algorithms. We have shown that the problem is NP-complete but is *bounded*. We have also developed *optimal* incremental violation detection algorithms for data partitioned vertically or horizontally, as well as optimization methods. Our experimental results have verified that these yield a promising solution to catching errors in distributed data.

There is naturally much more to be done. First, we are currently experimenting with real-life datasets from different applications, to find out when incremental detection is most effective. Second, we also intend to extend our algorithms to data that is partitioned both vertically and horizontally. Third, we plan to develop MapReduce algorithms for incremental violation detection. Fourth, we are to extend our approach to support constraints defined in terms of similarity predicates (*e.g.*, matching dependencies for record matching) beyond equality comparison, for which hash-based indices may not work and more robust indexing techniques need to be explored.

Acknowledgments. This work is supported in part by a RSE-NSFC Joint Project grant and an IBM scalable data analytics for a smarter planet innovation award. Fan, Li, Tang and Yu are also supported in part by NSFC 61133002, and the 973 Program 2012CB316200 of China.

References

[1] MD5. *http://en.wikipedia.org/wiki/MD5*.
[2] S. Agrawal, S. Deb, K. V. M. Naidu, and R. Rastogi. Efficient detection of distributed constraint violations. In *ICDE*, 2007.
[3] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.
[4] J. Bailey, G. Dong, M. Mohania, and X. S. Wang. Incremental view maintenance by base relation tagging in distributed databases. *Distributed and Parallel Databases*, 6(3), 1998.
[5] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1), 1981.
[6] J. A. Blakeley, P. Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, 1986.
[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[8] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In *SIGMOD*, 2007.
[9] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.
[10] W. Fan, F. Geerts, S. Ma, and H. Müller. Detecting inconsistencies in distributed data. In *ICDE*, 2010.
[11] W. Fan, J. Li, N. Tang, and W. Yu. Incremental detection of inconsistencies in distributed data. In *ICDE*, 2012. Available at *http://homepages.inf.ed.ac.uk/s0949090/icde12.pdf*.
[12] P. W. Frey, R. Goncalves, M. L. Kersten, and J. Teubner. A spinning join that does not get dizzy. In *ICDCS*, 2010.
[13] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
[14] A. Gupta and I. S. Mumick. *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.
[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.
[16] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *SIGMOD*, 1993.
[17] N. Huyn. Maintaining global integrity constraints in distributed databases. *Constraints*, 2(3/4), 1997.
[18] R. Kallman et al. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 2008.
[19] A. Kementsietsidis, F. Neven, D. Craen, and S. Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. In *VLDB*, 2008.
[20] D. Kossman. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4), 2000.
[21] J. Li, A. Deshpande, and S. Khuller. Minimizing communication cost in distributed multi-query processing. In *ICDE*, 2009.
[22] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, 1986.
[23] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, 2008.
[24] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing across multiple queries in MapReduce. *PVLDB*, 2010.
[25] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.
[26] G. Ramalingam and T. W. Reps. A categorized bibliography on incremental computation. In *POPL*, 1993.
[27] G. Ramalingam and T. W. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1&2), 1996.
[28] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *TODS*, 16(3), 1991.
[29] M. Stonebraker et al. C-store: A column-oriented DBMS. In *VLDB*, 2005.
[30] X. Wang, R. C. Burns, A. Terzis, and A. Deshpande. Network-aware join processing in global-scale database federations. In *ICDE*, 2008.

Wenfei Fan is Professor (Chair) of Web Data Management in the School of Informatics, University of Edinburgh, UK. He is a Fellow of the Royal Society of Edinburgh, UK, a National Professor of the Thousand-Talent Program and a Yangtze River Scholar, China. He received his PhD from the University of Pennsylvania, and his MS and BS from Peking University. He is a recipient of the Alberto O. Mendelzon Test-of-Time Award of ACM PODS 2010, the Best Paper Award for VLDB 2010, the Roger Needham Award in 2008 (UK), the Best Paper Award for ICDE 2007, the Outstanding Overseas Young Scholar Award in 2003, the Best Paper of the Year Award for Computer Networks in 2002, and the Career Award in 2001 (USA). His current research interests include database theory and systems, in particular data quality, data integration, distributed query processing, query languages, recommender systems, social networks and Web services.

Jianzhong Li is a professor and the chairman of the Department of Computer Science and Engineering at the Harbin Institute of Technology, China. He worked in the University of California at Berkeley as a visiting scholar in 1985. From 1986 to 1987 and from 1992 to 1993, he was a scientist in the Information Research Group in the Department of Computer Science at Lawrence Berkeley National Laboratory, USA. He was also a visiting professor at the University of Minnesota at Minneapolis, Minnesota, USA, from 1991 to 1992 and from 1998 to 1999. His current research interests include database management systems, data warehousing, data mining, and wireless sensor networks. He has authored three books and published more than 200 papers in refereed journals and conference proceedings, such as VLDB Journal, Algorithmic, IEEE Transactions on Knowledge and Data Engineering, IEEE Transactions on Parallel and Distributed Systems, Parallel and Distributed Database, SIGMOD, VLDB, ICDE, INFOCOM, ICDCS. He has been involved in the program committees of major computer science and technology conferences, including SIGMOD, VLDB, ICDE, INFOCOM, ICDCS, and WWW. He has also served on the editorial boards for distinguished journals, including Knowledge and Data Engineering, and refereed papers for varied journals and proceedings.

Nan Tang is a research scientist at QCRI (Qatar Computing Research Institute), Qatar Foundation, Qatar. He received the PhD degree from the Chinese University of Hong Kong in 2007. He has worked as a research staff member at CWI, the Netherlands, from 2008 to 2010. He was a research fellow at University of Edinburgh, from 2010 to 2012. His current research interests include data quality and graph database management.

Wenyuan Yu is a PhD student at University of Edinburgh, UK, working with Prof. Wenfei Fan. His research interests include data quality and social graph query.